Modelo de Computación Conexionista Inspirado en las Redes de Procesadores Evolutivos y su Aprendizaje

Autor: Miguel Angel Díaz Martínez
Director: Luis Fernando de Mingo López

17 de abril de 2009

Índice general

Resumen

La informática teórica es una disciplina básica ya que la mayoría de los avances en informática se sustentan en un sólido resultado de esa materia. En los últimos años debido tanto al incremento de la potencia de los ordenadores, como a la cercanía del límite físico en la miniaturización de los componentes electrónicos, resurge el interés por modelos formales de computación alternativos a la arquitectura clásica de von Neumann. Muchos de estos modelos se inspiran en la forma en la que la naturaleza resuelve eficientemente problemas muy complejos. La mayoría son computacionalmente completos e intrínsecamente paralelos. Por este motivo se les está llegando a considerar como nuevos paradigmas de computación (computación natural). Se dispone, por tanto, de un abanico de arquitecturas abstractas tan potentes como los computadores convencionales y, a veces, más eficientes: alguna de ellas mejora el rendimiento, al menos temporal, de problemas NP-completos proporcionando costes no exponenciales. La representación formal de las redes de procesadores evolutivos requiere de construcciones, tanto independientes, como dependientes del contexto, dicho de otro modo, en general una representación formal completa de un NEP implica restricciones, tanto sintácticas, como semánticas, es decir, que muchas representaciones aparentemente (sintácticamente) correctas de casos particulares de estos dispositivos no tendrían sentido porque podrían no cumplir otras restricciones semánticas. La aplicación de evolución gramatical semántica a los NEPs pasa por la elección de un subconjunto de ellos entre los que buscar los que solucionen un problema concreto.

En este trabajo se ha realizado un estudio sobre un modelo inspirado en la biología celular denominado redes de procesadores evolutivos [55, 53], esto es, redes cuyos nodos son procesadores muy simples capaces de realizar únicamente un tipo de mutación puntual (inserción, borrado o sustitución de un símbolo). Estos nodos están asociados con un filtro que está definido por alguna condición de contexto aleatorio o de pertenencia. Las redes están formadas a lo sumo de seis nodos y, teniendo los filtros definidos por una pertenencia a lenguajes regulares, son capaces de generar todos los lenguajes enumerables recursivos independientemente del grafo subyacente. Este resultado no es sorprendente ya que semejantes resultados han sido documentados en la

literatura. Si se consideran redes con nodos y filtros definidos por contextos aleatorios –que parecen estar más cerca a las implementaciones biológicas– entonces se pueden generar lenguajes más complejos como los lenguajes no independientes del contexto. Sin embargo, estos mecanismos tan simples son capaces de resolver problemas complejos en tiempo polinomial. Se ha presentado una solución lineal para un problema NP-completo, el problema de los 3-colores.

Como primer aporte significativo se ha propuesto una nueva dinámica de las redes de procesadores evolutivos con un comportamiento no determinista y masivamente paralelo [55], y por tanto todo el trabajo de investigación en el área de la redes de procesadores se puede trasladar a las redes masivamente paralelas. Por ejemplo, las redes masivamente paralelas se pueden modificar de acuerdo a determinadas reglas para mover los filtros hacia las conexiones. Cada conexión se ve como un canal bidireccional de manera que los filtros de entrada y salida coinciden. A pesar de esto, estas redes son computacionalmente completas. Se pueden también implementar otro tipo de reglas para extender este modelo computacional. Se reemplazan las mutaciones puntuales asociadas a cada nodo por la operación de *splicing*. Este nuevo tipo de procesador se denomina procesador *splicing*. Este modelo computacional de Red de procesadores con *splicing* $ANSP$ *es semejante en cierto modo a los sistemas distribuidos en tubos de ensayo basados en splicing.*

Además, se ha definido un nuevo modelo [56] –Redes de procesadores evolutivos con filtros en las conexiones– , en el cual los procesadores tan sólo tienen reglas y los filtros se han trasladado a las conexiones. Dicho modelo es equivalente, bajo determinadas circunstancias, a las redes de procesadores evolutivos clásicas. Sin dichas restricciones el modelo propuesto es un superconjunto de los NEPs clásicos. La principal ventaja de mover los filtros a las conexiones radica en la simplicidad de la modelización.

Otras aportaciones de este trabajo ha sido el diseño de un simulador en Java [54, 52] para las redes de procesadores evolutivos propuestas en esta Tesis.

Sobre el término "procesador evolutivo" empleado en esta Tesis, el proceso computacional descrito aquí no es exactamente un proceso evolutivo en

el sentido Darwiniano. Pero las operaciones de reescritura que se han considerado pueden interpretarse como mutaciones y los procesos de filtrado se podrían ver como procesos de selección. Además, este trabajo no abarca la posible implementación biológica de estas redes, a pesar de ser de gran importancia.

A lo largo de esta tesis se ha tomado como definición de la medida de complejidad para los ANSP, una que denotaremos como tamaño (considerando tamaño como el número de nodos del grafo subyacente). Se ha mostrado que cualquier lenguaje enumerable recursivo L puede ser aceptado por un ANSP en el cual el número de procesadores está linealmente acotado por la cardinalidad del alfabeto de la cinta de una máquina de Turing que reconoce dicho lenguaje L. Siguiendo el concepto de ANSP universales introducido por Manea [65], se ha demostrado que un ANSP con una estructura de grafo fija puede aceptar cualquier lenguaje enumerable recursivo. Un ANSP se puede considerar como un ente capaz de resolver problemas, además de tener otra propiedad relevante desde el punto de vista práctico: Se puede definir un ANSP universal como una subred, donde sólo una cantidad limitada de parámetros es dependiente del lenguaje. La anterior característica se puede interpretar como un método para resolver cualquier problema NP en tiempo polinomial empleando un ANSP de tamaño constante, concretamente treinta y uno. Esto significa que la solución de cualquier problema NP es uniforme en el sentido de que la red, exceptuando la subred universal, se puede ver como un programa; adaptándolo a la instancia del problema a resolver, se escogerán los filtros y las reglas que no pertenecen a la subred universal. Un problema interesante desde nuestro punto de vista es el que hace referencia a como elegir el tamaño optimo de esta red

Summary

This thesis deals with the recent research works in the area of Natural Computing –bio-inspired models–, more precisely Networks of Evolutionary Processors first developed by Victor Mitrana and they are based on P Systems whose father is Georghe Paun. In these models, they are a set of processors connected in an underlying undirected graph, such processors have an object multiset (strings) and a set of rules, named evolution rules, that transform objects inside processors[55, 53],. These objects can be sent/received using graph connections provided they accomplish constraints defined at input and output filters processors have. This symbolic model, non deterministic one (processors are not synchronized) and massive parallel one[55] (all rules can be applied in one computational step) has some important properties regarding solution of NP-problems in lineal time and of course, lineal resources. There are a great number of variants such as hybrid networks, splicing processors, etc. that provide the model a computational power equivalent to Turing machines.

The origin of networks of evolutionary processors (NEP for short) is a basic architecture for parallel and distributed symbolic processing, related to the Connection Machine as well as the Logic Flow paradigm, which consists of several processors, each of them being placed in a node of a virtual complete graph, which are able to handle data associated with the respective node. All the nodes send simultaneously their data and the receiving nodes handle also simultaneously all the arriving messages, according to some strategies. In a series of papers one considers that each node may be viewed as a cell having genetic information encoded in DNA sequences which may evolve by local evolutionary events, that is point mutations. Each node is specialized just for one of these evolutionary operations. Furthermore, the data in each node is organized in the form of multisets of words (each word appears in an arbitrarily large number of copies), and all the copies are processed in parallel such that all the possible events that can take place do actually take place. Obviously, the computational process just described is not exactly an evolutionary process in the Darwinian sense. But the rewriting operations we have considered might be interpreted as mutations and the filtering process might be viewed as a selection process. Recombination is missing but it was

asserted that evolutionary and functional relationships between genes can be captured by taking only local mutations into consideration.

It is clear that filters associated with each node allow a strong control of the computation. Indeed, every node has an input and output filter; two nodes can exchange data if it passes the output filter of the sender and the input filter of the receiver. Moreover, if some data is sent out by some node and not able to enter any node, then it is lost. In this paper we simplify the ANSP model considered in by moving the filters from the nodes to the edges. Each edge is viewed as a two-way channel such that the input and output filters coincide.

Clearly, the possibility of controlling the computation in such networks seems to be diminished. For instance, there is no possibility to loose data during the communication steps. In spite of this and of the fact that splicing is not a powerful operation (remember that splicing systems generates only regular languages) we prove here that these devices are computationally complete. As a consequence, we propose characterizations of two complexity classes, namely NP and PSPACE, in terms of accepting networks of restricted splicing processors with filtered connections.

We proposed a uniform linear time solution to SAT based on ANSPFCs with linearly bounded resources. This solution should be understood correctly: we do not solve SAT in linear time and space. Since any word and auxiliary word appears in an arbitrarily large number of copies, one can generate in linear time, by parallelism and communication, an exponential number of words each of them having an exponential number of copies. However, this does not seem to be a major drawback since by PCR (Polymerase Chain Reaction) one can generate an exponential number of identical DNA molecules in a linear number of reactions.

It is worth mentioning that the ANSPFC constructed above remains unchanged for any instance with the same number of variables. Therefore, the solution is uniform in the sense that the network, excepting the input and output nodes, may be viewed as a program according to the number of variables, we choose the filters, the splicing words and the rules, then we assign all possible values to the variables, and compute the formula.

We proved that ANSP are computationally complete. Do the ANSPFC remain still computationally complete? If this is not the case, what other problems can be efficiently solved by these ANSPFCs? Moreover, the complexity class NP is exactly the class of all languages decided by ANSP in polynomial time. Can NP be characterized in a similar way with ANSPFCs?

Parte I

Introducción

Capítulo 1

Computación Natural

1.1. Perspectiva Histórica

El afán del hombre para resolver situaciones con las que se ha encontrado a lo largo de la historia le ha llevado a la búsqueda de métodos o algoritmos de resolución que le permitan resolver en tiempo y espacio todo tipo de problemas. Es bien sabido que la Naturaleza y los procesos que en ella se realizan han sido una fuente constante de inspiración para encontrar estos algoritmos. Sin embargo, es en la segunda mitad del siglo XX, cuando se han realizado muchos esfuerzos por comprender los cálculos que la naturaleza realiza y utilizar este conocimiento para conseguir mejores algoritmos e incluso nuevos tipos de computadores. Ejemplos bien conocidos de la aplicación del conocimiento obtenido de la biología son las redes de neuronas artificiales y los algoritmos genéticos o, de forma más general, la computación evolutiva. Todos estos modelos inspirados en la naturaleza se han agrupado bajo la denominación de Computación Natural. El alcance y la importancia de estos modelos han quedado demostrados en su utilización en Inteligencia Artificial para abordar problemas de reconocimiento de patrones y otros a los que se ha encontrado soluciones satisfactorias. En este contexto, en los últimos diez o doce años, han surgido dos nuevos modelos computacionales que se agrupan bajo la etiqueta de Computación molécular: La computación con ADN

[66, 23] y la Computación con Membranas [82]. La importancia de estos modelos no convencionales de computación para la Sociedad de la Información y las Comunicaciones es manifiesta ya que son modelos masivamente paralelos que proporcionan una gran capacidad de computación y pueden abordar la solución de problemas NP-completos en tiempos polinómicos, llegando en algunos casos a tiempos lineales.

Se piensa que la investigación entre la frontera de la biología y la física con las teconologias de la información puede llevar al desarrollo de nuevos e importantes sistemas de información y tecnologías de computación. La cuestión crucial consiste en cómo podemos aprender de los sistemas biológicos y físicos y cómo podemos adaptarlos para desarrollar las tecnologías de la información del futuro. La computación biomolécular y la computación cuántica persiguen estos objetivos. El desarrollo tecnológico actual está permitiendo manipular de forma cada vez más precisa la materia a nivel molécular e incluso atómico. Estos avances pueden hacer realidad estos nuevos modelos de computación.

En estos modelos, las capacidades de procesamiento de información de las moléculas orgánicas pueden ser utilizadas en los computadores para reemplazar los componentes digitales inorgánicos actuales. Dentro de este ámbito se enmarca lo que se conoce con el nombre de Computación Natural, en el que se encuentran actualmente dos tipos muy novedosos de computación: la computación con ADN y la computación con membranas. Ambos proponen modelos computacionales inspirados en la estructura del ADN y las células vivas respectivamente.

La computación con Membranas extrae sus principios de las células vivas. En el modelo básico, los sistemas de membranas –también denominados Transition P Systems–, son modelos de computación no deterministas, masivamente paralelos y distribuidos, en los que se procesan multiconjuntos de objetos (moléculas), de forma sincronizada en los compartimentos delimitados por una estructura de membranas. Los objetos, que se corresponden con compuestos químicos, evolucionan en los compartimentos de la célula, pudiendo atravesar las membranas del sistema. Las membranas determinan una estructura jerarquizada y pueden ser disueltas, divididas y creadas, y su

permeabilidad puede ser alterada. Basados en este modelo han surgido múltiples variantes: membranas activas [86, 33, 35, 11, 75, 38], Tissue P Systems [28, 98, 99, 63, 44], Symport/antipor [20, 19, 36], NEP's [12, 13], etc.

La computación con membranas comenzó a finales de 1998 con la aportación de Gherorghe Paun, al publicar un artículo que circuló por la Web y que finalmente fue publicado en el año 2000 en Journal of Computer and System Science. En 1999 se crea en Leiden (Holanda) el Consorcio de Computación molécular Europeo (EMCC) con el objetivo de promover la investigación, especialmente en Europa, en Computación con ADN y Computación con Membranas "`http://openit.disco.unimib.it/emcc`*". El Grupo de Computación Natural de la Universidad Politécnica de Madrid es miembro fundador del EMCC. Bajo el patrocinio del EMCC se celebran actualmente, con carácter anual encuentros entre los diferentes grupos europeos que trabajan en dicha área.*

Desde el punto de vista social y de innovación tecnológica cabe destacar la capacidad de diferentes variantes de las Redes de Procesadores Evolutivos para resolver en tiempo polinomial problemas NP-completos. Estas variantes están basadas en la capacidad de las células para dividirse [4, 93]. Las posibilidades teóricas que ofrece esta característica permiten pensar en que diferentes problemas, que hasta hoy en día son intratables con la tecnología actual, puedan pasar a ser resueltos obteniendo resultados de alta aplicabilidad en la Sociedad de la Información y las Comunicaciones. A modo de ejemplo, son notorios los trabajos en los que se abordan la solución de problemas NP completos clásicos como son: el problema de la mochila, el problema del camino hamiltoniano en un grafo [41, 15, 70, 9, 62, 71, 68], el problema del viajante de comercio, etc. Los beneficios directos de su utilización son evidentes, ya que han sido problemas complejos. En los últimos dos años, las contribuciones en congresos internacionales y workshops organizados en torno a esta área de la computación han marcado un punto de inflexión en el alcance hacia la aplicación del modelo a problemas biológicos y sociales de difícil resolución con las técnicas tradicionales.

Con todo esto, se puede afirmar que el modelo formal y teórico está perfectamente recogido en la literatura existente. Y su aplicación está reconocida

en diversas áreas de investigación de fuerte impacto en la sociedad de la información actual.

La estrecha interacción entre la computación con ADN [96, 30] y la computación con membranas [87, 64], aparentemente diferentes, se puede apreciar en la forma en la que se establece en Europa el Consorcio de Computación molécular Europeo (EMCC). Este consorcio europeo nace en 1999 con el objetivo de promover la investigación sobre estos dos novedosos modelos de computación como se puede ver en su declaración de constitucion:

> *El consorcio de computación molécular europeo ha sido creado para coordinar, incentivar y expandir la investigación en este campo novedoso y excitante, especialmente en Europa. El EMCC es el resultado de las discusiones entre diferentes grupos de investigación en doce países europeos diferentes. Una clave del consorcio es incentivar la cooperación entre socios científicos, técnicos e industriales. Un sincero esfuerzo se realizará para crear cooperación multidisciplinar genuina entre las Ciencias de la Computación, la Biología molécular, y otras áreas relevantes. El EMCC organiza varias actividades de potenciación de la investigación como: conferencias, workshops, escuelas y visitas mutuas que dotan de foros científicos para el intercambio de resultados, y para establecer o fortalecer cooperaciones existentes en el campo de la computación molécular.*

Al igual que en la naturaleza, los modelos de Computación Natural están basados o en su parte viva (Biocomputación [69]) o en su parte inerte (como por ejemplo la técnica de Simulated Annealing [26, 34]).

Los modelos de Biocomputación se pueden clasificar por niveles de agregación de la forma siguiente:

1. *Nivel de molécula - Computación con ADN*
2. *Nivel de célula - Computación con membranas*
3. *Nivel de órgano - Computación con redes de neuronas*

Esta clasificación está basada en la propia Biología. Considerando sociedades de individuos en cualquiera de estos tres niveles, podemos introducir conceptos tan interesantes como el de selección natural. Estos conceptos aportan nuevos paradigmas capaces de resolver problemas computacionales [33].

La computación con membranas se puede encuadrar perfectamente en este marco. Estamos hablando de modelos de computación [17, 84, 85, 21], no de simulaciones de células vivas. Lo interesante de estos modelos es su capacidad para la resolución de problemas computacionales. La computación con membranas es un campo muy reciente de la computación molécular.

1.1.1. Computación con ADN

Un trabajo muy citado en el campo de la computación molécular, es el realizado por L. M. Addleman [80, 3, 2], publicado en 1994. En su trabajo describe el cómputo molécular de soluciones de problemas de combinatoria. En particular, el experimento logró resolver el problema de la ruta Hamiltoniana para una pequeña cantidad de nodos. Adleman resolvía en un laboratorio de biología molécular este problema matemático complejo utilizando un tubo de ensayo con ADN y aplicándole ciertas técnicas. Este trabajo supuso un gran avance en las ciencias de la computación que ya mostraba que era posible realizar cómputos a nivel molécular y, además, con una enorme capacidad de paralelismo inherente.

El estudio de cómo las caracteristicas genéticas pasan de generación en generación ha despertado el interés de muchos científicos y les ha incitado a descubrir la naturaleza de los genes. A principios de los años 40 se llega al descubrimiento de que el material hereditario se encuentra en los cromosomas y que está formado por ácido desoxirribonucleico (ADN) y proteínas. Pero no es hasta principios de los 50 cuando se resuelve la duda de cual de las dos sustancias es el material hereditario, llegándose a la conclusión que es el ADN quien lleva la información genética.

En 1953 James Watson y Francis Crick presentaron su modelo de ADN de doble hélice. En él se describía la composición del ADN como una doble

hélice, donde cada una de ellas estaba a su vez compuesta por una cadena de aminoácidos. Este estudio les supuso el Premio Nobel. El modelo de doble hélice permite explicar las propiedades que se esperan del ADN.

Adleman observó que en la mitosis de la célula interviene una enzima, la polimerasa. Al fijarse en cómo esta enzima creaba una copia perfecta de la información genética, se maravilló con el parecido que tenía ésta en su forma de actuar con una máquina de Turing de doble cinta. Inmediatamente se puso a trabajar en la elaboración de un experimento que utilizara el ADN para realizar cálculos computacionales. En 1994 publicaría este experimento en una revista científica [2], en el que resolvía una instancia del problema del camino Hamiltoniano.

Con los resultados de Adleman se ha despertado un gran interés en lo que hoy se conoce como cómputo con ADN. Esta es una rama de la biología computacional, que sirve como intersección entre las ciencias computacionales, las matemáticas y la biología [43, 74]. La clave de lo que hace especialmente interesante este modelo, con respecto a un modelo de computación tradicional, está en la capacidad para el paralelismo masivo. Este paralalismo permite que en un tubo se estén realizando millones de operaciones simultáneas por segundo.

1.1.2. Computación con membranas

La computación con membranas se puede encuadrar en el marco de modelos de biocomputación a nivel de célula. Esto significa que estamos hablando de modelos con capacidad para la resolución de problemas computacionales. A continuación se presenta, por un lado, el sistema biológico en el que se inspira este modelo teórico. Por otro, se caracteriza formalmente este modelo y se presentan sus principales cualidades en cuanto a su potencia desde el punto de vista de la computación.

El funcionamiento de la célula ha sido implementado en el modelo de la computación celular con membranas, los Sistemas P. De forma resumida, lo que se denomina objetos de este modelo, corresponden a los orgánulos que

hay en el citoplasma. Estos, mediante las señales que reciben, consecuencia de las reacciones químicas, pueden transformarse dentro del citoplasma. Estas transformaciones internas y el paso selectivo hacia dentro y hacia fuera que permite la membrana a algunos orgánulos, corresponden a las reglas internas que en este modelo se describen.

Los sistemas de computación con membranas son modelos de computación basados en los procesos biomoléculares de los seres vivos. Las investigaciones se apoyan en la idea de que la imitación de los procedimientos que tienen lugar en la Naturaleza y su aplicación en las máquinas, nos puede conducir a descubrir y desarrollar nuevos modelos de computación que darán lugar a una nueva generación de ordenadores con mayor capacidad de proceso.

Estos sistemas se pueden describir como una estructura jerárquica de membranas que delimitan unas regiones, en las que se sitúan multiconjuntos de objetos que evolucionan de acuerdo a reglas de evolución asociadas a cada región. Los objetos también se pueden comunicar de una región a otra de acuerdo a ciertas indicaciones. Asimismo, las membranas se pueden disolver, quedando los objetos contenidos en esa membrana libres en la región inmediatamente externa. También se pueden dividir y el contenido de cada membrana que se divide se replica en cada una de las membranas resultantes. Por último, las reglas de evolución deben aplicarse de forma masivamente paralela; esto significa que, en cada unidad de tiempo, todos los objetos que puedan evolucionar, deben evolucionar.

1.1.3. Computación con redes de neuronas

Con las redes de neuronas artificiales se busca la solución de problemas complejos, no como una secuencia de pasos, sino como la evolución de unos sistemas de computación inspirados en el cerebro humano, y dotados por tanto de cierta inteligencia. Estos sistemas de computación son una combinación de elementos simples de proceso interconectados, que operando de forma paralela en varios estilos, consiguen resolver problemas relacionados con el reconocimiento de formas o patrones [47, 48], predicción [46, 51, 49], codificación, control y optimización [45], entre otras aplicaciones.

La teoría y modelado de redes neuronales está inspirada en la estructura y funcionamiento de los sistemas nerviosos, donde la neurona es el elemento fundamental. En general, una neurona consta de un cuerpo celular más o menos esférico del que salen una rama principal, el axón, y varias ramas más cortas, llamadas dendritas. Una de las características de las neuronas es su capacidad para comunicarse.

En términos generales las dendritas y el cuerpo celular reciben señales de entrada; el cuerpo celular las combina e integra y emite señales de salida. El axón transmite dichas señales a los teminales axónicos, que distribuyen información a un nuevo conjunto de neuronas, se calcula que en el cerebro humano existen del orden de 10^{15} *conexiones. Las señales que se utilizan son de dos tipos: eléctrica y química. La señal generada por la neurona y transportada a lo largo del axón es un impulso eléctrico, mientras que la señal que se transmite entre los terminales axónicos de una neurona y las dendritas de la otra es de origen químico.*

Para establecer una similitud directa entre la actividad sináptica y la analogía con las redes de neuronas artificiales podemos considerar que las señales que llegan a la sinapsis son las entradas a la neurona; éstas son ponderadas a través de un parámetro, denominado peso [47]. Estas señales de entrada pueden excitar a la neurona o inhibirla. El efecto es la suma de las entradas ponderadas. Si la suma es igual o mayor que el umbral de la neurona, entonces la neurona se activa. Esta es una situación de todo o nada; cada neurona se activa o no se activa. La facilidad de transmisión de señales se altera mediante la actividad del sistema nervioso. Las sinapsis son susceptibles a la fatiga, deficiencia de oxígeno y la presencia de anestésicos, entre otros. Esta habilidad de ajustar señales es un mecanismo de aprendizaje.

Las redes de neuronas artificiales (RNA) son modelos que intentan reproducir el comportamiento del cerebro. Como tal modelo, realiza una simplificación, averiguando cuales son los elementos relevantes del sistema, bien porque la cantidad de información de que se dispone es excesiva o bien porque es redundante. Una elección adecuada de sus caracteristicas, más una estructura conveniente, es el procedimiento convencional utilizado para construir redes capaces de realizar una determinada tarea [50].

1.2. Estructura y funciones de la célula

Una célula tiene una estructura compleja, con varios compartimentos delimitados dentro de la membrana principal por varias membranas internas: el núcleo, el aparato de Golgi, diferentes vesículas, etc. Todas estas membranas son parecidas, y actúan como separadores y filtros. En este apartado se tendrá en cuenta aquellos elementos de la célula relativos a la estructura y funcionamiento de la membrana que encierra el plasma celular [39, 30, 22].

El modelo aceptado actualmente de la estructura de membrana fue propuesto por S. Singer y G. Nicolson en 1972. Este modelo se conoce con el nombre de modelo del mosaico fluido. Según éste, una membrana es un fosfolípido de dos capas en el que moléculas proteicas (y otras tales como colesterol, esteroides, etc.), se encuentran total o parcialmente embebidas.

Las moléculas fosfolípidas están formadas principalmente por dos partes, una cabeza polarizada y una cola no polarizada [6, 5, 27, 61, 76].

La cabeza está formada por un grupo fosfato y un grupo nitrogenado, la cola está constituida por dos canales de ácidos grasos y está unida a la cabeza por un glicerol. Esta composición química es importante porque las cabezas de las moléculas en las dos capas son hidrofílicas, mientras que las colas son hidrofóbicas. Esto explica la distribución de cabezas limitando, tanto el plasma celular de la región interna, como la región externa a la célula. También explica la dificultad del paso de agua a través de la membrana. Más aún, la polaridad de las cabezas lleva a tener polaridades diferentes en las dos partes de la membrana: carga positiva en la capa externa y negativa en la capa interna de las moléculas. Esta polarización facilita la salida de iones negativos y la entrada de iones positivos.

El modelo de una membrana, se denomina mosaico fluido porque las moléculas fosfolípidas pueden moverse entre las dos capas, pero a causa de la polarización permanecen siempre entre los planos de las dos capas. Esto hace posible el movimiento de proteínas y de otros compuestos, lo que es importante desde el punto de vista de las comunicaciones intercelulares y extracelulares.

La membrana que delimita el plasma celular es parcialmente permeable. Por ejemplo, pequeñas moléculas neutras, desde el punto de vista de la carga, pueden atravesar una membrana, casi libremente, si son lípidos solubles. Moléculas de mayor tamaño pueden cruzarla de forma asistida únicamente, mientras que los iones cargados pasan selectivamente de una región a otra.

La transferencia de moléculas a través de la membrana puede darse de dos formas [8, 7, 60, 79]: de una forma pasiva, especialmente por difusión hacia la región de menor concentración, y de una forma activa (o mediatizada).

La transferencia activa de moléculas más importantes se realiza a través de los canales de proteínas, que se encuentran presentes de forma numerosa en las membranas vivas. Por ejemplo, el agua y ciertas macromoléculas pueden pasar a través de dichos canales.

En realidad hay principalmente dos tipos de canales de proteínas, los que seleccionan los objetos que los atraviesan por su tamaño, y los denominados portadores de proteínas que interactúan con moléculas específicas, llegando incluso en algunos casos a modificarlas cuando las ayudan a atravesarlos.

Otras funciones importantes de las proteínas de la membrana son: la actividad catalítica, el reconocimiento y el enlace. Todos estos detalles han sido considerados en la definición formal de los Sistemas P (especialmente en la consideración en el plano de estructuras de membranas); son la forma en la que células vecinas establecen canales de proteínas para la comunicación entre ellas. Debido al hecho de que la membrana es un mosaico fluido, cuando dos membranas se tocan, sus proteínas se buscan entre si; cuando dos proteínas se encuentran suficientemente próximas, se enlazan y esto potencia la formación de otros canales de proteínas similares. De esta forma se puede establecer una red compleja de comunicaciones entre células. Es considerablemente interesante que cuando una célula es invadida por moléculas no deseadas, sus canales con las células vecinas se cierran, y cuando se dan circunstancias favorables son abiertos de nuevo.

1.3. Bioquímica y Modelo Formal

Este es el momento de efectuan algunas observaciones sobre la relación entre los elementos del modelo y la realidad bioquímica.

En primer lugar, debe quedar claro que el objetivo de la computación con membranas no es construir un modelo de célula real, sino proponer un modelo computacional teórico inspirado en la estructura de la célula. En segundo lugar, las membranas que se consideran tienen dos propósitos, servir de separadores de objetos y de canales de comunicación. Cualquier clase de membrana, real o virtual, puede desempeñar estos dos papeles, el modelo de membrana es más bien un modelo que tiene un significado matemático y no necesariamente un significado bioquímico. Más aún, las membranas tienen un comportamiento pasivo, en contraste con las membranas biológicas. Como última cuestión, las estructuras lípidas de dos capas dejan pasar a través de ellas ciertos componentes químicos por razones de: concentración, gradiente, polarización eléctrica; o de forma mucho más selectiva a través de ciertos canales de proteínas. Esta última forma de comunicación entre membranas es similar a la comunicación con comandos de la forma (a, in_j)*. Las membranas reales pueden ser disueltas [83, 81], pero también pueden ser inhibidas y hacerse opacas a cualquier clase de comunicación.*

El modelo evolutivo de los Sistemas P [77, 10, 59] está basado en una interpretación bioquímica muy idealizada de la célula. Tenemos una célula, delimitada por la piel (la membrana de la célula). Dentro, hay órganos de la célula y moléculas libres, organizados jerárquicamente. Las moléculas y los órganos flotan aleatoriamente en el líquido citoplasmático de cada membrana. Bajo condiciones específicas, las moléculas evolucionan, solas o con la ayuda de ciertos catalizadores; éstos no son modificados en las reacciones que se producen (las reglas de evolución codifican las reacciones químicas entre los objetos que evolucionan juntos). Esto se produce en paralelo, síncronamente para todas las moléculas (se supone que existe un reloj universal). Las nuevas moléculas pueden permanecer en la misma región en la que aparecieron o pueden pasar a través de las membranas que delimitan este espacio, selectivamente. Algunas reacciones no sólo modifican moléculas, sino que también pueden romper las membranas. Cuando una membrana se rompe, las molécu-

las situadas en ella permanecen libres en el espacio en que se sumergen, pero las reglas de evolución de la membrana rota se pierden. La hipótesis que sustenta este hecho es que las condiciones de reacción de la membrana rota se pierden y en el espacio en que se sumergen sólo son válidas las reglas específicas para él. Si la membrana externa se pierde, la célula cesa su actividad y sus componentes se separan.

Las reacciones químicas que se producen en la célula se pueden asociar a reglas situadas en las regiones internas que las membranas definen. Debido a que cada una de las reglas se corresponde con una reacción química, la prioridad asociada a la regla se correspondería con la probabilidad de que la reaccion se produzca (algunas entradas químicas pueden ser más activas que otras) [81, 72]. El modelo de sistema P interpreta la probabilidad en un sentido muy restrictivo; si una regla con una prioridad mayor [92] se utiliza, entonces ninguna regla de menor probabilidad puede ser utilizada, incluso si no compiten por objetos. La interpretación que se da aquí es la correspondiente con la forma de utilizar prioridades en gramáticas ordenadas en el área de reescritura regulada, pero también tiene un significado bioquímico, si se piensa que cada regla no sólo consume objetos sino que también consume energía; si una regla de mayor prioridad se utiliza, entonces no queda energía para reglas de prioridad menor.

1.4. Redes de procesadores Evolutivos (NEPs)

1.4.1. Introducción

Empezaremos por describir los conceptos utilizados en esta Tesis[24].

- *Alfabeto: Conjunto finito y no vacío de símbolos.*
- *Cadena o palabra sobre V: Es cualquier secuencia de símbolos de un alfabeto V*
- *V^*: Conjunto de todas las cadenas sobre V*

- ϵ: *Cadena vacía*
- $Card(A)$: *Cardinalidad de un conjunto finito* A
- $|x|_a$: *Longitud de una cadena* x
- $alph(x)$: *Como alfabeto mínimo* W *tal que* $x \in W^*$ *[18, 36] para cada cadena no vacía* x

Además tendremos los siguientes tipos de reglas:

- *Regla de sustitución, cuando* $a \to b$, *con* $a, b \in V \cup \{\epsilon\}$ *si ambos* a *y* b *no son iguales a* ϵ.
- *Regla de Borrado, cuando* $a \neq \epsilon$ *y* $b = \epsilon$.
- *Regla de Inserción, cuando* $a = \epsilon$ *y* $b \neq \epsilon$.

El conjunto de todas las reglas de sustitución, inserción y borrado sobre un alfabeto V *se denota por* Sub_V, Del_V *e* Ins_V *respectivamente [16, 13].*

Dada una regla σ *y una cadena* $w \in V^*$, *se definen las siguientes acciones de* σ *sobre* w:

- *Si* $\sigma \equiv a \to b \in Sub_V$, *entonces:*

$$\sigma^*(w) = \sigma^l(w) = \sigma^r(w) = \begin{cases} \{ubv : \exists u, v \in V^*(w = uav)\} \\ \{w\}, \text{ en otro caso} \end{cases}$$

- *Si* $\sigma \equiv a \to \epsilon \in Del_V$, *entonces:*

$$\sigma^*(w) = \begin{cases} \{uv : \exists u, v \in V^*(w = uav)\} \\ \{w\}, \text{ en otro caso} \end{cases}$$

$$\sigma^l(w) = \begin{cases} \{v : w = av\} \\ \{w\}, \text{ en otro caso} \end{cases}$$

$$\sigma^r(w) = \begin{cases} \{u : w = ua\} \\ \{w\}, \ en\ otro\ caso \end{cases}$$

- *Si* $\sigma \equiv \epsilon \rightarrow a \in Ins_V$, *entonces:*

$$\sigma^*(w) = \{uav : \exists u, v \in V^*(w = uv)\}$$

$$\sigma^l(w) = \{wa\}$$

$$\sigma^r(w) = \{aw\}$$

$\alpha \in \{*, l, r\}$ *expresa la forma de aplicar una regla de evolución a una cadena, en cualquier posición* ($\alpha = *$), *por la izquierda* ($\alpha = l$) *o por la derecha* ($\alpha = r$). *Para cada regla* σ, *acción* $\alpha \in \{*, l, r\}$ *y* $L \subseteq V^*$ *se define la* α*-acción de* M *sobre la cadena* w *y el lenguaje* L *como:*

$$M^\alpha(w) = \bigcup_{\alpha \in M} \sigma^\alpha(w) \tag{1.1}$$

$$M^\alpha(L) = \bigcup_{w \in L} M^\alpha(w) \tag{1.2}$$

Las operaciones de reescritura [59, 78, 58, 63] definidas anteriormente se pueden denominar operaciones evolutivas ya que se ven como formulaciones lingüísticas de las mutaciones locales en los genes. Para dos subconjuntos disjuntos P *y* F *de un alfabeto* V *y una cadena sobre* V, *se definen los predicados:*

$$\begin{aligned} &\varphi^{(1)}(w; P, F) \equiv P \subseteq alph(w) \wedge F \cap alph(w) = \emptyset \\ &\varphi^{(2)}(w; P, F) \equiv alph(w) \subseteq P \\ &\varphi^{(3)}(w; P, F) \equiv P \subseteq alph(w) \wedge F \not\subseteq alph(w) \end{aligned} \tag{1.3}$$

La construcción de estos predicados está basada en condiciones de contexto aleatorio definido por los dos conjuntos P *(contexto permisivo) y* F *(contexto prohibitivo).*

Para todo lenguaje $L \subseteq V^*$ *y* $\beta \in \{(1),(2),(3)\}$ *se define:*

$$\varphi^{\beta}(L,P,F) = \{w \in L \mid \varphi^{\beta}(w;P,F)\}$$

1.4.2. Redes de procesadores evolutivos (NEPs)

Damos por suficientemente conocidos los conceptos fundamentales sobre gramáticas y autómatas finitos por lo que sólo incidiremos en alguno de ellos como: El conjunto de todas las cadenas sobre V *se denota* V^* *y la cadena vacía se representa por* ϵ*. Un multiconjunto sobre un conjunto* X *es una función* $M : X \to N \cup \{\infty\}$*. El número* $M(x)$ *expresa el número de copias de* $x \in X$ *en el multiconjunto* M*. Cuando* $M(x) = \infty$*, entonces* x *aparece arbitrariamente un gran número de veces en* M*. El conjunto* $supp(M)$ *es el soporte de* M*,* $supp(M) = \{x \in X | M(x) > 0\ \}$*. Para más información sobre estos conceptos ver[89].*

Un NEP de tamaño n *es una tupla* $\Gamma = (V, N_1, \cdots, N_n, G)$*, donde:*

- V *es un alfabeto.*
- $N_i = (M_i, A_i, PI_i, PO_i)\ \forall i = 1, \cdots, n$ *es el procesador evolutivo* i *asociado a la red de procesadores.*

Los parámetros de cada procesador son los siguientes:

- M_i *es un conjunto finito de reglas de evolución.*
  - $a \to b;\ a, b \in V$ *reglas de sustitución.*
  - $a \to \epsilon;\ a \in V$ *reglas de borrado.*
  - $\epsilon \to a;\ a \in V$ *reglas de inserción.*

  El conjunto de reglas de evolución de cualquier procesador contiene reglas de sustitución, borrado o inserción.

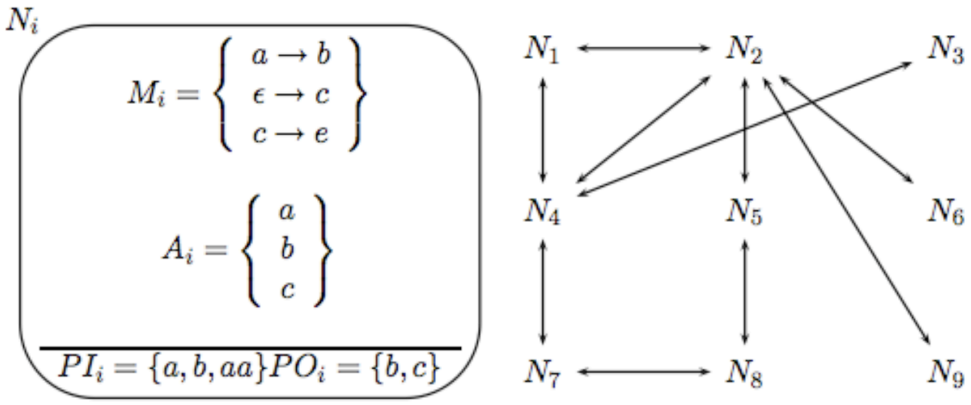


Figura 1.1: Red de procesadores evolutivos.

- *A_i es un conjunto finito de cadenas sobre V. El conjunto A_i es el conjunto inicial de cadenas en el procesador i. Se considera que cada cadena que aparece en un procesador determinado y en un momento concreto dispone de un número infinito de copias.*

- *PI_i y PO_i son subconjuntos de V que representan los filtros de entrada y salida respectivamente. Estos filtros se definen por la propiedad de pertenencia, esto es, una cadena $w \in V$ puede atravesar el filtro de entrada (salida) si $w \in PI_i$ ($w \in PO_i$).*

Finalmente $G = (\{N_1, \cdots, N_n\}, E)$ es un grafo no dirigido llamado grafo subyacente de la red, véase figura 1.1.

Una configuración (estado) de una red de procesadores evolutivos definida anteriormente es una n-tupla $C = (L_1, \cdots, L_n)$, con $L_i \subseteq V^*$ para todo $1 \leq i \leq n$. Una configuración representa el conjunto de cadenas (cada cadena aparece un gran número arbitrario de veces), que están presentes en un nodo en un momento determinado. La configuración inicial de la red es $C_0 = \{A_1, \cdots, A_n)$. Una configuración puede cambiar por una fase de evolución o por una fase de comunicación. Cuando cambia por una fase de evolución, cada compoenente L_i de la configuración se cambia de acuerdo a las reglas asociadas al nodo i.

Formalmente, se dice que la configuración $C_1 = (L_1 \cdots, L_n)$ cambia directamente hacia la configuración $C_2 = (L'_1, \cdots, L'_n)$ por una fase de evolución, denotado por $C_1 \Rightarrow C_2$ si L'_i es el conjunto de cadenas obtenido aplicando las reglas de R_i a las cadenas presentes en L_i como sigue:

- *Si las reglas de borrado o sustitución pueden reemplazar ocurrencias del mismo símbolo dentro de una cadena, entonces todas estas ocurrencias se deben reemplazar dentro de las diferentes copias de la cadena. El resultado es un multiconjunto en el que todas las cadenas que se pueden obtener aplicando estas reglas aparecen un gran número de veces.*
- *Una regla de inserción se aplica en cualquier posición de la cadena. El resultado es un multiconjunto en el que cualquier cadena que se puede obtener aplicando una regla de inserción en cualquier posición de una cadena existente aparece un gran número de veces.*
- *Si más de una regla, no importa de que tipo, se puede aplicar a una cadena entonces todas las reglas se emplean usando diferentes copias de la cadena.*

En otras palabras, ya que un gran número arbitrario de copias de cada cadena está disponible en un nodo, después de una fase de evolución en cada nodo se obtiene un gran número de copias de la cadena que se puede conseguir empleando cualquier regla del conjunto de reglas de evolución asociadas a ese nodo. Por definición, si L_i esta vacío para algún $1 \leq i \leq n$, entonces L'_i estará también vacío.

Cuando se produce un cambio por una fase de comunicación, cada nodo N_i envía todas las copias de las cadenas que son capaces de atravesar su filtro de salida hacia todos los nodos conectados a N_i y recibe todas las copias de las cadenas enviadas por cualquier otro procesador conectado con N_i siempre y cuando puedan atravesar su filtro de entrada. Formalmente se dice que la configuración $C_1 = (L_1, \cdots, L_n)$ cambia en la configuración $C_2 = (L'_1, \cdots, L'_n)$ por una fase de comunicación, denotado por $C_1 \vdash C_2$ si:

$$L_i' = L_i - \{w|w \in L_i \cap PO_i\} \cup \bigcup_{\{N_i,N_j\}\in E} \{x|x \in L_j \cap PO_j \cap PI_i\} \qquad (1.4)$$

Sea $\Gamma = (V, N_1, \cdots, N_n)$ *una red de procesadores evolutivos definida anteriormente. Una computación en* Γ *es una secuencia de configuraciones* $C_0, C_1, \cdots$, *donde* C_0 *es la configuración inicial y* $C_{2i} \Rightarrow C_{2i+1}$ *y* $C_{2i+1} \vdash C_{2i+2}$ *para todo* $i \geq 0$.

Si la secuencia es finita, entonces se consigue una computación finita. El resultado de cualquier computación finita o infinita es un lenguaje que se recoge en un nodo designado, llamado nodo de salida (master) de la red. Si se considera el nodo de salida de la red como el nodo N_k, *y si* $C_0, C_1, \cdots$ *es una computación, entonces todas las cadenas existentes en el nodo* k *en algún instante* t *(el* k*-ésimo componente de* C_t*), pertenecen al lenguaje generado por la red. Esto se denota por* $L_k(\Gamma)$.

La complejidad para computar un conjunto finito de cadenas Z *es el mínimo número de fases* t *en una computación* $C_0, C_1, \cdots, C_t, \cdots$ *tal que* Z *es un subconjunto de la* k*-ésima componente de* C_t.

Completitud computacional

Teorema 1.4.1. *Los lenguajes recursivamente enumerables se pueden generar empleando un NEP completo de tamaño* 5 *[13].*

Demostración. Sea $G = (N, T, S, P)$ una gramática arbitraria en la forma normal de Kuroda [42], y P contiene unicamente reglas de los siguientes tipos:

$$A \to a, A \to BC, AB \to CD, A \to \epsilon$$

donde A, B, C, D son no terminales y a es un terminal. Se asume que

las reglas $A \to BC$ y $AB \to CD$ de P están etiquetadas de manera unívoca por las etiquetas $r_1, r_2, \cdots, r_n$. Nos referiremos a las reglas de la forma $A \to a, A \to \epsilon, A \to BC$, y $AB \to CD$ como reglas del tipo $0, 1, 2$ y 3 respectivamente. Se contruye el siguiente NEP de tamaño 5 sobre un grafo completo subyacente:

$$\Gamma = (N \cup T \cup V \cup \{X\}, N_0, N_1, N_2, N_3, N_4, K_5)$$

donde $V = \{r_i, p_i, q_i, s_i, t_i | 1 \leq i \leq n\}$ y

$$\begin{aligned}
N_0 &= (\emptyset, \emptyset, T^*, (N \cup T \cup V \cup \{X\})^*(N \cup V \cup \{X\})(N \cup T \cup V \cup \{X\})^*) \\
N_1 &= (M_1, \{S\}, (N \cup T)^* \cup (N \cup T)^*\{r_i q_i | 1 \leq i \leq n\}(N \cup T)^* \\
&\quad (N \cup T)^*(\{r_i, s_i p_i, t_i q_i | 1 \leq i \leq n\} \cup \{X\})(N \cup T)^* \cup T^*) \\
\text{con} &\quad M_1 = \{A \to X | A \to \epsilon \in P\} \cup \{A \to a | A \to a \in P\} \cup \{A \to r_i, \\
&\quad r_i \to t_i | r_i : A \to BC \in P\} \cup \{A \to s_i, B \to p_i | r_i : AB \to CD \in P\} \\
N_2 &= (\{\epsilon \to q_i | r_i : A \to BC\}, \emptyset, (N \cup T)^*\{r_i | 1 \leq i \leq n\}(N \cup T)^*, \\
&\quad (N \cup T \cup V)^*) \\
N_3 &= (M_3, \emptyset, (N \cup T)^*\{s_i p_i, t_i q_i | 1 \leq i \leq n\}(N \cup T)^*, (N \cup T)^*) \\
\text{con} &\quad M_3 = \{t_i \to B, q_i \to C | r_i : A \to BC \in P\} \cup \\
&\quad \{s_i \to C, p_i \to D | r_i : AB \to CD \in P\} \\
N_4 &= (\{X \to \epsilon\}, \emptyset, (N \cup T)^*\{X\}(N \cup T)^*, (N \cup T)^*)
\end{aligned}$$

A continuación se detallan unas explicaciones del funcionamiento de esta red. Inicialmente, existe un número grande de copias de las cadena S en el nodo N_1. Por extrapolación, se puede asumir que un multiconjunto de cadenas contiene como poco un símbolo no terminal y ningún símbolo de $V \cup \{X\}$, cada cadena que aparece en el multiconjunto tiene un número de copias no acotado, está presente en N_1 en un momento dado –antes de cualquier paso de evolución–.

Si existe una cadena x en el multiconjunto que tiene como mínimo una ocurrencia de un no terminal A y $A \rightarrow Y$ es una regla de evolución en M_1, entonces la primera ocurrencia de A en x se reemplaza por Y con un número infinito de copias de x, y así para todas las ocurrencias de A en x. Este proceso se aplica a todas las cadenas en N_1 para todas la reglas de evolución (sustitución) en M_1.

Aquellas cadenas que se han obtenido por la aplicación de la regla $A \rightarrow r_i$, para algún i, (esto quiere decir que existe una regla $r_i : A \rightarrow BC$ en P), se envían fuera y se reciben por N_2 o N_4 que son los únicos nodos capaces de recibirlas. Aquellas cadenas que contienen X se reciben por N_4 y las otras por N_2. En N_4, el símbolo X se elimina y las cadenas obtenidas se envían de vuelta a N_1, siempre y cuando contengan símbolos no terminales, o hacia N_0 si son cadenas terminales.

Por tanto, todas las posibles aplicaciones de las reglas de tipo 1 se realizan en cuatro pasos. En N_2 se inserta q_j en cualquier posición de las cadenas existentes de la misma manera que se ha descrito anteriormente para la regla $A \rightarrow Y$, para todos j tal que r_j sea una regla de tipo 2.
Sólo aquellas cadenas que tengan una ocurrencia de r_i que reciban un símbolo adyacente q_i a la derecha de r_i pueden abandonar el nodo N_2, todas las demás permanecerán en N_2 para siempre. Las cadenas que abandonan N_2 no se pueden recibir más que por el nodo N_1 donde las únicas reglas de evolución que se pueden aplicar son aquellas de la forma $r_i \rightarrow t_i$. Después de aplicar estas reglas, las nuevas cadenas se envían fuera de nuevo. Todas las otras reglas que se aplican a las cadenas recibidas en N_1 conllevan nuevas cadenas que permacecerán en N_0 para siempre. Las cadenas enviadas son recibidas por N_3 donde t_i y q_i son reemplazadas por B y C, respectivamente, siempre y cuando la parte derecha de la regla r_i sea BC.
De este modo, la red realiza todas posibles aplicaciones de las reglas de tipo 2 en diez pasos.

De manera semejante, la red realiza la aplicación de las reglas de tipo 3 en ocho pasos. Si se desea aplicar la regla $r_i : AB \rightarrow CD$, en N_1, en un paso evolutivo, una ocurrencia de A se reemplaza por s_i, estas nuevas cadenas permanecen en N_1 durante la siguiente fase de comunicación ya que

no puede atravesar el filtro de salida de N_1, en la siguiente fase de evolución una ocurrencia de B se reemplaza por p_i, pero sólo en aquellas cadenas que contienen la subpalabra $s_i p_i$ (esto quiere decir que una subpalabra AB fue reemplazada por $s_i p_i$), puede atravesar el filtro de salida de N_1, las otras permanecerán en N_1 para siempre.

Como se puede observar, las cadenas sobre el alfabeto $N \cup T$ siempre vuelven a N_1, donde el proceso anteriormente comentado comienza para todas las que todavía contienen símbolos no terminales, mientras que las cadenas terminales se envían a N_0 donde permanecen para siempre.

Por tales explicaciones, se puede inferir que el lenguaje generado por Γ en el nodo de salida N_0 es exactamente $L(G)$. □

Vale la pena reseñar que, a diferencia de otros dispositivos paralelos generadores de lenguajes, un NEP genera el lenguaje de una manera muy eficiente. Todas las cadenas que se pueden generar por una gramática, cada una en n pasos, se generan conjuntamente en un NEP en $10n$ pasos como mucho.

Si se presta atención a la demostración, se puede ver que se puede modificar el NEP completo por un NEP estrella siendo N_1 el nodo central, y por tanto se puede decir:

Teorema 1.4.2. *Los lenguajes recursivamente enumerables se pueden generar empleando un NEP con un grafo de estrella de tamaño 5 [13].*

Una situación semejante ocurre para NEPs en anillo, esto es:

Teorema 1.4.3. *Los lenguajes recursivamente enumerables se pueden generar empleando un NEP con un grafo de anillo de tamaño 6 [13].*

Demostración. La construcción del NEP es bastante similar a la del teorema 1.4.1. Para la misma gramática de la anterior demostración se considera un NEP de tamaño 6:

$$\Gamma = (N \cup T \cup U \cup \{X\}, N_0, N_1, N_2, N_3, N_4, N_5, G)$$

donde $U = \{r_i, p_i, q_i, s_i | 1 \neq i \leq n\}$ y

$$
\begin{aligned}
N_0 &= (\emptyset, \emptyset, T^*, (N \cup T \cup U)^*(N \cup U)(N \cup T \cup U)^*), \\
N_1 &= (M_1, \{S\}, (N \cup T)^* \cup (N \cup T)^*\{r_i, s_i p_i | 1 \leq i \leq n\}(N \cup T)^* \cup T^*), \\
\text{con} \quad & M_1 = \{A \to X | A \to \epsilon \in P\} \cup \{A \to a | A \to a \in P\} \cup \{A \to r_i, \\
& | r_i : A \to BC \in P\} \cup \{A \to s_i, B \to p_i | r_i : AB \to CD \in P\} \\
N_2 &= (\{\epsilon \to q_i | r_i : A \to BC\}, \emptyset, (N \cup T)^*\{r_i, s_i p_i | 1 \leq i \leq n\}(N \cup T)^*, \\
& (N \cup T)^*\{s_i p_i, r_i q_i | 1 \leq i \leq n\}(N \cup T)^*) \\
N_3 &= (M_3, \emptyset, (N \cup T)^*\{s_i p_i, r_i q_i | 1 \leq i \leq n\}(N \cup T)^*, (N \cup T)^*, \\
& (N \cup T)^*\{r_i q_i | 1 \leq i \leq n\}(N \cup T)^*), \\
\text{con} \quad & M_3 = \{s_i \to C, p_i \to D | r_i : AB \to CD \in P\} \\
N_4 &= (M_4, \emptyset, (N \cup T)^*\{r_i q_i | 1 \leq i \leq n\}(N \cup T)^*, (N \cup T)^*), \\
\text{con} \quad & M_4 = \{r_i \to B, q_i \to C | r_i : A \to BC \in P\}, \\
N_5 &= (\{X \to \epsilon\}, \emptyset, (N \cup T)^*(\{X\} \cup \{r_i, s_i p_i | 1 \leq i \leq n\})(N \cup T)^*, \\
& (N \cup T \cup U)^*).
\end{aligned}
$$

El gráfo subyacente es un anillo con 6 nodos. Siguiendo la anterior demostración del teorema 1.4.1 se puede concluir que $L(G) = L_0(\Gamma)$. □

A continuación se mencionan algunos resultados que muestran como se pueden emplear las redes de procesadores evolutivos para resolver el tipo de problemas NP-completos en tiempo lineal.

Teorema 1.4.4. *Bounded Post Correspondence Problem (BPCP) se puede resolver empleando un NEP acotado linealmente en tiempo y tamaño por el producto de K y la longitud de las dos listas Post [12].*

BPCP es una variante del famoso problema de computación denominado Post Correspondence Problem (PCP) que se con la tecnología actual es difícil de abordar hoy en día [29]. Una instancia del PCP consiste en un alfabeto V y dos listas de cadenas sobre V.

$$u = (u_1, \cdots, u_n) \text{ y } v = (v_1, \cdots, v_n) \tag{1.5}$$

El problema consiste en responder si es cierto o no que existe una secuencia de números enteros positivos $i_1, \cdots, i_k$, cada uno entre 1 y n, tal que:

$$u_{i_1}, \cdots, u_{i_k} = v_{i_1}, \cdots, v_{i_k} \tag{1.6}$$

El problema es indecidible cuando no existe un límite superior para k y NP-completo si k está acotado por una constante $K \leq n$. Una solución al problema BPCP basada en ADN se propone en [40].

Demostración. Sean $u = (u_1, u_2, \cdots, u_n)$ y $v = (v_1, v_2, \cdots, v_n)$ las dos listas Post sobre el alfabeto $V = \{a_1, a_2, \cdots, a_m\}$ y $K \geq n$. Sea:

$$s = Kmax(\{|u_j| \mid 1 \leq j \leq n\} \cup \{|v_j| \mid 1 \leq j \leq n\}) \tag{1.7}$$

Consideresé un nuevo alfabeto:

$$U = \bigcup_{i=1}^{m} \{a_i^{(1)}, a_i^{(2)}, \cdots, a_i^{(s)}\} = \{b_1, b_2, \cdots, b_{sm}\} \tag{1.8}$$

Para cada $x = i_1 i_2 \cdots 1_j \in \{1, 2, \cdots, n\} \leq K$ (el conjunto de todas las secuencias de longitud como mucho K formado por enteros entre 1 y n), se define la cadena:

$$u^{(x)} = u_{i_1} u_{i_2} \cdots u_{i_j} = a_{t_1} a_{t_2} \cdots a_{t_{p(x)}} \tag{1.9}$$

Ahora se define una función uno a uno: $\alpha : V^* \to U^*$ tal que para cada secuencia x citada anteriormente $\alpha(u^{(x)})$ no contiene dos ocurrencias del mismo símbolo de U. Se puede tomar:

$$\alpha(u^{(x)}) = a_{t_1}^{(1)} a_{t_2}^{(2)} \cdots a_{t_{p(x)}}^{(p(x))} \tag{1.10}$$

La misma construcción se puede aplicar a las cadenas que se encuentran en la segunda lista Post v. Se define:

$$F = \{\alpha(u^{(x)}\alpha(v^{(x)}) \mid x \in \{1, 2, \cdots, n\} \leq K\} = \{z_1, z_2, \cdots, z_l\} \tag{1.11}$$

y se asume que $z_j = b_{j,1} b_{j,2} \cdots b_{j,r_j}$, $1 \leq j \leq l$, donde $|z_j| = r_j$. Tal y como se ha construído F, no existe ninguna letra de U que aparezca dentro de ninguna cadena en F más de dos veces. Además, si cada letra que aparece en $z = \alpha(u^{(x)})\alpha(v^{(x)})$ aparece dos veces en z, entonces x representa una solución para la instancia.

Ahora se puede definir la Red de Procesadores Evolutivos que calcula todas las soluciones de la instancia dada. Es un NEP de tamaño $2sm + 1$.

$$\Gamma = (U \cup \bar{U} \cup \hat{U} \cup \tilde{U} \cup \{X\} \cup \{X_d^{(c)} \mid 1 \leq c \leq n, 2 \leq d \leq |z|_c\}, N_1, N_2, \cdots, N_{2sm+1})$$

donde,

$$\begin{aligned} \bar{U} &= \{\bar{b} \mid b \in U\} \\ \hat{U} &= \{\hat{b} \mid b \in U\} \\ \hat{U} &= \{\hat{b} \mid b \in U\} \end{aligned}$$

Para todo $1 \leq f \leq sm$,

$$
\begin{aligned}
M_f &= \{\epsilon \to b_f\} \\
A_f &= \emptyset \\
FI_f &= \{X_d^{(c)} \mid 2 \leq d \leq |z|_c, 1 \leq c \leq l \text{ tal que } b_f \neq b_{c,d}\} \cup \bar{U} \cup \hat{U} \cup \tilde{U} \\
PI_f &= FO_f = PO_f = \emptyset
\end{aligned}
$$

y

$$
\begin{aligned}
M_{sm+1} &= \{X \to X_2^{(c)} \mid 1 \leq c \leq l\} \cup \{X_{|z|_c} \to b_{c,1}\} \cup \{b_d \to \bar{b}_d \mid 1 \leq d \leq sm\} \\
&\quad \cup \{X_c^{(c)} \to X_{d+1}^{(c)} \mid 1 \leq c \leq l, 2 \leq d \leq |z|_c - 1\} \\
A_{sm+1} &= \{X\} \\
FI_{sm+1} &= PI_{sm+1} = FO_{sm+1} = PO_{sm+1} = \emptyset
\end{aligned}
$$

Además, para todo $1 \leq d \leq sm$

$$
\begin{aligned}
M_{sm+d+1} &= \{b_d \to \tilde{b}, \bar{b}_d \to \hat{b}_d\} \\
A_{sm+d+1} &= \emptyset \\
FI_{sm+d+1} &= (\bar{U} \setminus \{\bar{b}_d\}) \cup \{X_g^{(c)} \mid 2 \leq g \leq |z|_c, 1 \leq c \leq l\} \\
PI_{sm+d+1} &= FO_{sm+d+1} = \emptyset \\
PO_{sm+d+1} &= \{\tilde{b}_d, \hat{b}_d\}
\end{aligned}
$$

A continuación se exponen algunas consideraciones informales sobre el modo de computación de este tipo de red de procesadores evolutivos. En la primera etapa de computación sólo los procesadores $1, 2, \cdots, sm+1$ están activos. Ya que el filtro de entrada de los otros procesadores contiene todos los símbolos de la forma $X_g^{(c)}$; éstos permanecen inactivos hasta que una cadena de F se genere en el nodo $sm+1$.

Primero se muestra como una cadena arbitraria $z_j = b_{j,1}b_{j,2}\cdots b_{j,r_j}$ de F se puede obtener en el nodo $sm+1$.

El proceso comienza aplicando las reglas $X \to X_2^{(j)},\ 1 \leq j \leq l$ en el

nodo $sm+1$. Las cadenas $X_2^{(j)},\ 1 \leq j \leq l$, obtenidas en el nodo $sm+1$ como consecuencia de una fase de evolución se envían a todos los otros procesadores, pero para cada una de estas cadenas sólo existe un procesador que las puede recibir. Por ejemplo la cadena $X_2^{(c)}$ es únicamente aceptada por el procesador $f,\ 1 \leq f \leq sm$, con $b_f = b_{c,2}$. En el siguiente paso de evolución, el símbolo $b_{j,2}$ se añade a la parte derecha de la cadena $X_2^{(j)}$ para todos $1 \leq j \leq l$. Ahora, se produce una fase de comunicación. Todas las cadenas $X_2^{(j)}b_{j,2}$ pueden atravesar los filtros de salida de los procesadores donde se han obtenido pero el procesador $sm+1$ es el único que puede recibirlas. Aquí los subíndices del símbolo X se incrementan en una unidad y el proceso de arriba se repite con el objetivo de añadir una nueva letra. Este proceso no se aplica a la cadena $X_r^{(j)}b_{j,2}\cdots b_{j,r}$ si y sólo si $r = |z_j|$, cuando $X_r^{(j)}$ se reemplaza por $b_{j,1}$ resultando la cadena z_j. Por dichas consideraciones, se puede inferir que todas las cadenas de F se producen en el nodo $sm+2$ en $2s$ pasos.

Otra tarea en la computación comprueba el número de ocurrencias de cualquier letra dentro de cualquier cadena obtenida en el nodo $sm+1$, siempre y cuando la cadena contenga solamente letras en U.
Esto es. Recordando la aplicación de las reglas de sustitución mencionada con anterioridad, cada ocurrencia de cualquier letra a se reemplaza por su versión $\bar{a}$ en el nodo $sm+1$. Considérese una cadena producida por tal fase de evolución. Tal cadena tiene únicamente una ocurrencia de un símbolo $\bar{b}_d$, para algún $1 \leq d \leq sm$, y los otros símbolos pertenecen a $U \cup \hat{U} \cup \tilde{U}$. Ésta puede atravesar únicamente el filtro de entrada del procesador $sm+d+1$, donde permanece durante tres fases (dos fases de evolución y una de comunicación) o para siempre.

La cadena puede abandonar el nodo $sm+d+1$ sólo si tiene una ocurrencia del símbolo b_d. Reemplazando esta ocurrencia por $\tilde{b}_d$ y $\bar{b}_d$ por $\hat{b}_d$, la cadena puede atravesar el filtro de salida del procesador $sm+d+1$ e ir al nodo $sm+1$. De esta manera se comprueba si la cadena original ha tenido dos ocurrencias de la letra b_d.

Después de 6 fases la computación finaliza y el nodo $sm+1$ tiene únicamente cadenas que fueron producidas a partir de aquellas cadenas en F que

tenían dos ocurrencias de cualquier letra. Estas cadenas codifican todas las soluciones de la instancia del *BPCP* dada. □

1.4.3. Redes de procesadores evolutivos simples

Un NEP simple es una variante de los NEPs, los parámetros son los mismos que en los NEPs exceptuando los filtros. En los NEPs simples los filtros están basados en condiciones de contexto arbitrarias.

Un NEP simple [13] de tamaño n es una tupla:

$$\Gamma = (V, N_1, N_2, \cdots, N_n, G)$$

donde, V y G representan el alfabeto y el grafo subyacente respectivamente, y para cada $1 \leq i \leq n, N_i = (M_i, A_i, PI_i, FI_i, PO_i, FO_i)$ *es el i-ésimo procesador evolutivo de la red.* M_i *es el conjunto finito de reglas de evolución y* A_i *es un conjunto finito de cadenas sobre V, pero:*

- PI_i *y* FI_i *son subconjuntos de* V *que representan el filtro de entrada. Este filtro viene definido por condiciones de contexto aleatorias,* PI_i *define la condición de contexto permisiva y* FI_i *define la condición de contexto prohibitiva.*

  Sea $w \in V^*$ *una cadena, ésta podrá atravesar el filtro de entrada del procesador i, si w contiene cada elemento de* PI_i *pero no contiene elementos de* FI_i*. Si una o varias condiciones de contexto aleatorias son el conjunto vacío las correspondientes comprobaciones de contexto son obviadas.*

  Diremos que $\rho_i(w) = true$*, si w puede atravesar el filtro de entrada del procesador i y* $\rho_i(w) = false$*, si no puede.*

- PO_i *y* FO_i *son subconjuntos de* V *que representan el filtro de salida. Una cadena puede atravesar el filtro de salida de un procesador si satisface las condiciones de contexto aleatorias asociadas al mismo.*

Diremos que $\tau_i(w) = true$, si w puede atravesar el filtro de salida del procesador i y $\tau_i(w) = false$, si no puede.

Teorema 1.4.5. *Las familias de lenguajes regulares y lenguajes de contexto libre son incomparables con la familia de lenguajes generados por las redes de procesadores evolutivos simples [13].*

Demostración. El número de todas las ocurrencias de un símbolo/letra a en una cadena x se denota por $|x|_a$. Inicialmente se muestra que el lenguaje $L = \{x \in \{a,b,c\}^* || x|_a = |x|_b = |x|_c \geq 1\}$ se puede generar empleando un NEP simple de tamaño 6. Para tal objetivo se considera el lenguaje $V = \{X, a, b, c, a', b'\}$ y se construye el NEP simple $\Gamma = (V, N_0, N_1, N_2, N_3, N_4, N_5)$, donde:

$$
\begin{aligned}
N_0 &= (\{\epsilon \to a', \epsilon \to X\}, \{\epsilon\}, \emptyset, \{a, b, c, X\}, \{X, a'\}, \emptyset), \\
N_1 &= (\{\epsilon \to b'\}, \emptyset, \{a'\}, \emptyset, \emptyset, \emptyset), \\
N_2 &= (\{a' \to a\}, \emptyset, \{a', b'\}, \emptyset, \emptyset, \emptyset), \\
N_3 &= (\{\epsilon \to c\}, \emptyset, \{b'\}, \{a'\}, \emptyset, \emptyset), \\
N_4 &= (\{b' \to b\}, \emptyset, \{b', c\}, \emptyset, \emptyset, \emptyset), \\
N_5 &= (\{X \to \epsilon\}, \emptyset, \{a, b, c\}, \{b'\}, \emptyset, \{a, b, c\}).
\end{aligned}
$$

Describiremos por encima el funcionamiento por el que este NEP genera las cadenas.
El nodo de inicio N_0 y después de $n+1$ pasos de evolución todas las cadenas que contienen n ocurrencias de a' y una ocurrencia de X se envían hacia el resto de nodos conectados con él. Siendo N_1 el único nodo capaz de recibirlas; en N_1 el símbolo b' se inserta en cualquier posible posición, resultando en cadenas que contienen n ocurrencias de a' y una ocurrencia de X y b', respectivamente. Después de un paso de comunicación, todas estas cadenas llegan a N_2, donde una ocurrencia de a' se reemplaza por a. Ahora, todas las cadenas se devuelven a N_1 y el proceso continúa. El proceso continúa hasta que todas las ocurrencias de a' son transformadas en a. Durante este proceso

el mismo número de ocurrencias, esto es n, de b' se insertan de tal manera que al final del proceso, sólo aquellas cadenas que tienen n ocurrencias de a y el mismo número de ocurrencias de b', junto con una ocurrencia de X existen en N_2.

Posteriormente este proceso se aplica a todas las ocurrencias de b' y c en los nodos N_3 y N_4. Cuando todas aquellas cadenas que tienen el mismo número, n, de ocurrencias de cada letra a, b, y c y todavía una ocurrencia de X, producidas en N_4, son recibidas por N_5, donde X se elimina de dichas cadenas y las cadenas resultantes permanecen para siempre. Por tanto $L = L_5(\Gamma)$.

El lenguaje regular $\{a^n, b^m | n, m \geq 1\}$ no se puede generar empleando un NEP simple. □

Los NEPs simples, podrían emplearse para resolver problemas NP completos, nos centramos en un problema conocido como es el problema de los 3 *colores.*
Este problema consiste en decidir como colorear cada uno de los vértices de un gráfo no dirigido empleando tres colores de tal manera que después de colorearlos no existan dos vértices conectados por una arista con el mismo color.

Teorema 1.4.6. *El problema de 3-colores se puede resolver en tiempo* $O(n+m)$ *empleando un NEP simple completo de tamaño* $7m + 2n$*, donde* n *es el número de vértices y* m *es el número de aristas del grafo de entrada [13].*

Demostración. Sea $G = (\{1, 2, \cdots, n\}, \{e_1, e_2, \cdots, e_m\})$ un gráfo y $e_t = \{i_t, j_t\}, 1 \leq i_t < j_t \leq n, 1 \leq t \leq m$. Se considera el alfabeto $U = V \cup V' \cup T \cup \{X_1, X_2, \cdots, X_{m+1}\}$, donde $V = \{b_1, r_1, g_1, \cdots, b_n, r_n, g_n\}$, $T = \{a_1, a_2, \cdots, a_n\}$. V' denota la copia prima de V, esto es el conjunto formado por las copias primas de todas las letras presentes en V. Se construye el siguiente NEP completo simple de tamaño $7m + 2$ teniendo los nodos:

$$
\begin{aligned}
N_0 &= (\{a_i \to b_i, a_i \to r_i, a_i \to g_i | 1 \leq i \leq n\}, \{a_1 a_2 \cdots a_n X_1\}, \\
&\quad T \cup \{X_1\}, \emptyset, \emptyset, T).
\end{aligned}
$$

El resto de los nodos son:

$$
\begin{aligned}
N_{e_t}^{(Z)} &= (\{Z_{i_t} \to Z'_{i_t}\}, \emptyset, \{X_t\}, U \setminus V, \{Z'_{i_t}\}, \emptyset), Z \in \{b, r, g\}, \\
\bar{N}_{e_t}^{(b)} &= (\{r_{j_t} \to r'_{j_t}, g_{j_t} \to g'_{j_t}\}, \emptyset, \{X_t, b'_{i_t}\}, \emptyset, \{r'_{j_t}, g'_{j_t}\}, \emptyset), \\
\bar{N}_{e_t}^{(r)} &= (\{b_{j_t} \to b'_{j_t}, g_{j_t} \to g'_{j_t}\}, \emptyset, \{X_t, r'_{i_t}\}, \emptyset, \{b'_{j_t}, g'_{j_t}\}, \emptyset), \\
\bar{N}_{e_t}^{(g)} &= (\{r_{j_t} \to r'_{j_t}, b_{j_t} \to b'_{j_t}\}, \emptyset, \{X_t, g'_{i_t}\}, \emptyset, \{r'_{j_t}, b'_{j_t}\}, \emptyset), \\
N_{e_t} &= (\{r'_{i_t} \to r_{i_t}, b'_{i_t} \to b_{i_t}, g'_{i_t} \to g_{i_t}, r'_{j_t} \to r_{j_t}, b'_{j_t} \to b_{j_t}, g'_{j_t} \to g_{j_t}, \\
&\quad X_t \to X_{t+1}\}, \emptyset, \{X_t\}, \{r_{i_t}, b_{i_t}, g_{i_t}, r_{j_t}, b_{j_t}, g_{j_t}\}, \{X_{t+1}\}, U \setminus V),
\end{aligned}
$$

para todo $1 \leq t \leq m$, y $N_1 = (\{X_{m+1} \to \epsilon\}, \emptyset, \{X_{m+1}\}, U \setminus V, \emptyset, V)$.

En los $2n$ primeros pasos, donde n son de comunicación nada se envía, las cadenas permanecen en N_0 hasta que aparezca alguna letra de T. Una vez finalizado este proceso, las cadenas resultantes codificarán todas las posibles maneras de colorear los vértices, satisfaciendo o no los requerimientos del problema.

Para cada arista e_t, el NEP sólo almacena aquellas cadenas que codifican una condición de coloreado para los dos vértices de e_t. Esto se consigue mediante los nodos $N_{e_t}^{(b)}, N_{e_t}^{(r)}, N_{e_t}^{(g)}, \bar{N}_{e_t}^{(b)}, \bar{N}_{e_t}^{(r)}, \bar{N}_{e_t}^{(g)}$, y finalmente N_{e_t} en 8 pasos. La configuración alcanza una solución si el lenguaje del nodo N_1 es no vacío.

Se puede deducir que el tiempo global de computación es $8m + 2n$ y que el número total de reglas es $16m + 3n + 1$. Además indicar que todos los parámetros de la red son de tamaño $O(m + n)$. □

El gráfo subyacente del NEP anterior no depende del número de nodos

de la instancia del problema. En otras palabras, se puede emplear la misma estructura para resolver el problema de los 3 *colores siempre y cuando se tenga el mismo número de aristas sin importar el número de nodos.*

La construcción anterior se puede modificar con el objetivo de comenzar con un número infinito de copias de la cadena vacía. Para tal objetivo, se necesita un alfabeto T' *con* n *símbolos extra* $a'_1, a'_2, \cdots, a'_n$ *y* $n+1$ *nodos extra:*

$$\begin{aligned} N'_0 &= (\{\epsilon \to a'_i | 1 \leq i \leq n\}, \{X_1\}, \emptyset, U \setminus T', T' \cup \{X_1\}, \emptyset), \\ N'_i &= (\{a'_i \to a_i\}, \emptyset, T_i \cup \{X_1\}, T' \setminus T_i, \emptyset, \emptyset), \end{aligned}$$

para todo $1 \leq i \leq n$. *Para cada* i, *se denota* $T_i = \{a_1, a_2, \cdots, a_{i-1}, a'_i, \cdots, a'_n\}$.

Esta vez, la computación comienza generando las cadenas que contienen todas las letras de T' *y ninguna ocurrencia de* X_1 *en el nodo* N'_0. *Esto consume* $2n$ *pasos. Todas estas cadenas se envían fuera y el único nodo capaz de recibirlas es* N'_1. *En los siguientes* $2n$ *pasos aquellas cadenas que inicialmente contenían exactamente una ocurrencia de cada símbolo de* T' *son recibidas, por turnos, en los nodos* $N'_1, N'_2, \cdots, N'_n$ *y finalmente* N_0. *Seguidamente la computación continúa de la misma manera que la demostración anterior.*

1.4.4. Redes híbridas de procesadores evolutivos(HNEPs)

Un procesador sobre V *esta formado por la tupla* (M, PI, FI, PO, FO):

- $(M \subseteq Sub_V)$ *o* $(M \subseteq Del_V)$ *o* $(M \subseteq Ins_V)$. *El conjunto* M *representa el conjunto de reglas evolutivas del procesador. Como se puede apreciar un procesador está especializado únicamente en una operación evolutiva.*
- $PI, FI \subseteq V$ *son los contextos permisivos/prohibitivos de entrada, mien-*

tras $PO, FO \subseteq V$ *son los contextos permisivos/prohibitivos de salida del procesador.*

Se denota por EP_V *al conjunto de procesadores evolutivos sobre* V*.*

Una red híbrida de procesadores evolutivos (HNEP) es una 7-tupla $\Gamma = (V, G, N, C_0, \alpha, \beta, i_0)$*, donde:*

- V *es un alfabeto.*
- $G = (X_G, E_G)$ *en un grafo no dirigido donde* X_G *es el conjunto de vértices y* E_G*. es el conjunto de aristas y al que denominaremos grafo subyacente de la red.*
- $N : X_G \to EP_V$ *es una función que asocia cada procesador evolutivo* $N(x) = (M_x, PI_x, FI_x, PO_x, FO_x)$ *con cada nodo* $x \in X_G$*.*
- $C_0 : X_G \to 2^{V^*}$ *es la función que identifica la configuración inicial de la red. Asociando un conjunto de cadenas finito a cada nodo del grafo* G*.*
- $\alpha : X_G \to \{*, l, r\}$*;* $\alpha(x)$ *indica el modo en el que se aplicarán las reglas del nodo* x *sobre las cadenas existentes en dicho nodo.*
- $\beta : X_G \to \{(1), (2), (3)\}$ *define el tipo de filtros de entrada/salida de un nodo. Para todo nodo,* $x \in X_G$*, se definen los siguientes filtros:*

$$\textit{filtro de entrada: } \rho_x(.) = \varphi^{\beta(x)}(.; PI_x, FI_x)$$

$$\textit{filtro de salida: } \tau_x(.) = \varphi^{\beta(x)}(.; PO_x, FO_x)$$

  $\rho_x(w)$ *(respectivamente* $\tau_x(w)$*) indicará si la cadena* w *puede atravesar el filtro de entrada (salida) del procesador* x*. Genéricamente,* $\rho_x(L)$ *(respectivamente* $\tau_x(L)$*) será el conjunto de cadenas de* L *que pueden atravesar el filtro de entrada (salida) del procesador* x*.*
- $i_0 \in X_G$ *es el nodo de salida de HNEP.*

Donde $card(X_G)$ *es el tamaño de* Γ. *Si* $\alpha(x) = \alpha(y)$ *y* $\beta(x) = \beta(y)$ *para cualquier par de nodos* $x, y \in X_G$, *diremos que la red es homogénea. Se mostrarán redes de procesadores evolutivos con sus grafos subyacentes de diferentes tipos como anillos, estrellas, mallas, etc. Así diremos que un HNEP es una estrella, anillo, malla o grafo completo si su grafo subyacente es una estrella, anillo, malla o un grafo completo. Los grafos de estrella, anillo, malla y grafos completos de n vértices se denotan por* S_n, R_n *y* K_n.

La función $C : X_G \to 2^{V^*}$ *que asocia un conjunto de cadenas con todos los nodo del grafo es una configuración de un HNE. Ésta se puede entender como todos los conjuntos de cadenas presentes en los nodos del grafo en un momento dado. Las configuraciones pueden cambiar por una fase de evolución y por una fase de comunicación. El cambio pr@oducido en una fase de evolución, de cada componente* $C(x)$ *de la configuración* C *vendrá determinado por el conjunto de reglas de evolución* M_x *asociadas al nodo* x *y será resultado de aplicar las reglas* $\alpha(x)$. *Diremos que una configuración* C' *se obtiene en una fase de evolución a partir de la configuración* C, *y denotaremos como* $C \Rightarrow C'$, *si y sólo si:*

$$C'(x) = M_x^{\alpha(x)}(C(x)) \forall x \in X_G$$

Si se produce un cambio en la configuración en una fase de comunicación, cada procesador $x \in X_G$ *envía una copia de cada cadena que posee, que es capaz de atravesar el filtro de salida de* x, *a todos los procesadores conectados a* x; *por otro lado recibe todas las cadenas enviadas por cualquier nodo conectado a* x *siempre que pueda atravesar su filtro de entrada.*

Diremos que una configuración C' *se obtiene por una fase de comunicación a partir de la configuración* C, *y denotaremos como* $C \vdash C'$, *si y sólo si:*

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))) \forall x \in X_G$$

Si Γ es un HNEP, la computación producida en Γ es una secuencia de configuraciones $C_0, C_1, \cdots$, donde C_0 es la configuración inicial de Γ, $C_{2i} \Rightarrow C_{2i+1}$ y $C_{2i+1} \vdash C_{2i+2}$ para todo $i \geq 0$. Donde, cada configuración C_i es unívocamente determinada por la configuración C_{i-1}. Si la secuencia es finita, se tiene una computación finita. Si utilizamos un HNEP como dispositivo de generación de lenguajes, entonces el resultado de una computación finita o infinita es un lenguaje que se obtendrá en el nodo de salida de la red.

Para cualquier computación $C_0, C_1, \cdots$, todas las cadenas existentes en el nodo de salida en cualquier momento pertenecerán al lenguaje generado por la red. Denotaremos el lenguaje generado por Γ como $L(\Gamma) = \bigcup_{s \geq 0} C_s(i_0)$.

Diremos que la compejidad para computar un conjunto finito de cadenas Z es el número mínimo s tal que $Z \subseteq \bigcup_{t=0}^{s} C_t(i_0)$.

HNEP como dispositivos generadores de lenguajes

Compararemos estos dispositivos con la gramáticas más simples en la jerarquía de Chomsky. En [13], se muestra que las familias de lenguajes regulares y lenguajes de contexto libre son incomparables con la familia de lenguajes generados por redes homogéneas de procesadores evolutivos. Los HNEP son más potentes realizando este trabajo.

Teorema 1.4.7. *Cualquier lenguaje regular se puede generar por cualquier tipo (estrella, anillo, grafo completo) de HNEP.*

Demostración. Sea $A = (Q, V, \delta, q_0, F)$ un autómata finito determinista; sin perder generalidad se puede asumir que $\delta(q, a) \neq q_0$ para cada $q \in Q$ y cada $a \in V$. Además se puede asumir que $card(V) = n$. Se construye el siguiente HNEP completo.

$$\Gamma = (U, K_{2n+3}, N, C_0, \alpha, \beta, f)$$

El alfabeto U se define como $U = V \cup V' \cup Q \cup \{s_a \mid s \in Q, a \in V\}$, donde $V' = \{a' \mid a \in V\}$. El conjunto de los nodos del grafo completo subyacente es

$Node$	M	PI	FI	PO	FO	C_0	α	β
x_0	$\{q \to s_b\}_{\sigma(s,b)=q}$	$\emptyset$	$\{s_b\} \cup \{b'\}_b$	$\emptyset$	$\emptyset$	F	$*$	(1)
$a \in V$	$\epsilon \to a'$	$\{s_a\}_s \cup V$	Q	U	$\emptyset$	$\emptyset$	l	(2)
$a' \in V'$	$\{s_a \to s\}_s$	$\{a'\}$	Q	$\emptyset$	$\emptyset$	$\emptyset$	$*$	(1)
x_1	$\{b' \to b\}_b$	$\emptyset$	$\{s_b\}_{s,b}$	$\emptyset$	$\emptyset$	$\emptyset$	$*$	(1)
x_f	$q_0 \to \epsilon$	$\{q_0\}$	V'	$\emptyset$	V	$\emptyset$	r	(1)

Tabla 1.1: Configuración del HNEP, véase [67]

$\{x_0, x_1, x_f\} \cup V \cup V'$, y el resto de parámetros vienen en la tabla 1.1, donde s y b son estados genéricos de Q y símbolos de V, respectivamente.

Se puede probar por inducción que:

1. $\delta(q, x) \in F$ para algún $q \in Q \setminus \{q_0\}$ si y sólo si $xq \in C_{8|x|}(0)$.
2. x es aceptada por $A(x \in L(A))$ si y sólo si $x \in C_p(f)$ para cualquier $p \geq 8|x| + 1$. Por tanto, $L(A)$ es exactamente el lenguaje generado por Γ.

El tamaño del HNEP, y su estructura subyacente, no depende del número de estados del autómata. En otras palabras, esta estructura es común a todos los lenguajes regulares sobre el mismo alfabeto, no importa la complejidad del autómata que lo reconoce. Además, todas las cadenas de la misma longitud se generan simultáneamente.

Cualquier gramática lineal se puede transformar en una gramática lineal equivalente con reglas del tipo $A \to aB, A \to Ba, A \to \epsilon$, la demostración del teorema se puede adaptar para probar el siguiente resultado. □

Teorema 1.4.8. *Cualquier lenguaje lineal se puede generar por un HNEP de cualquier tipo.*

Teorema 1.4.9. *Existen lenguajes que no son de contexto libre y se pueden generar por un HNEP de cualquier tipo.*

Demostración. Se construye un HNEP completo que genera los lenguajes que no son de contexto libre $L = \{wcx \mid x \in \{a, b\}^*, w \text{ es una permutación de } x\}$.

$Node$	M	PI	FI	PO	FO	C_0	α	β
y_0	$\{\epsilon \to X, \epsilon \to D\}$	$\emptyset$	$\{a', b', a, b, X_a, X_b\}$	$\{D\}$	$\emptyset$	$\{\epsilon\}$	r	(1)
y_1	$\{\epsilon \to X_a, \epsilon \to X_b\}$	$\emptyset$	$\{X_a, X_b, a', b'\}$	$\emptyset$	$\emptyset$	$\emptyset$	r	(1)
y_u	$\{X \to u'\}$	$\{X_u\}$	$\{a', b'\}$	$\emptyset$	$\emptyset$	$\emptyset$	$*$	(1)
$\bar{y}_u$	$\{X_u \to u\}_b$	$\{u'\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$*$	(1)
$\tilde{y}_u$	$\{u' \to u\}$	$\{u'\}$	$\{X_a, X_b\}$	$\emptyset$	$\emptyset$	$\emptyset$	$*$	(1)
y_2	$\{D \to c\}$	$\emptyset$	$\{X, a', b', X_a, X_b\}$	$\emptyset$	$\{a, b\}$	$\emptyset$	$*$	(1)

Tabla 1.2: Configuración inicial del HNEP, véase [67]

$$\Gamma = (V, K_9, N, C_0, \alpha, \beta, y_2)$$

donde $V = \{a, b, a', b', X_a, X_b, X\}$, $X_{K_9} = \{y_0, y_1, y_2, y_a, y_b, \bar{y}_a, \bar{y}_b, \tilde{y}_a, \tilde{y}_b\}$, y el resto de parámetros vienen especificados en la tabla 1.2, donde u es un símbolo genérico de $\{a, b\}$.

El funcionamiento de esta red es bastante simple. En el nodo y_0 existen cadenas generadas con la forma X^n para cualquier $n \geq 1$. Éstas abandonan este nodo tan pronto como reciban una D en su extremo derecho, sea y_1 el único nodo capaz de aceptarlo. En y_1, o X_a o X_b se añaden al final derecho. Así, para un n dado, las cadenas X^nDX_a y X^nDX_b se producen en y_1. Donde las cadenas X^nDX_a, de forma similar se puede aplicar a las cadenas X^nDX_b, X^nDX_a se dirige hacia y_a y cualquier ocurrencia de X se reemplaza por a' en el mismo número de copias que X^nDX_a. y_a produce cada cadena $X^ka'X^{n-k-1}DX_a, 0 \leq k \leq n-1$. Todas las cadenas son envíadas fuera del procesador, pero solo $\bar{y}_a$, puede recibirlas. Donde, X_a es reemplazada por a y las cadenas obtenidas se envían a $\tilde{y}_a$ y la a es reemplazada por a'. Tanto en cuanto las cadenas contengan ocurrencias de X, éstas siguen el mismo camino, $y_1, y_u, \bar{y}_y, \tilde{y}_u,\ u \in \{a, b\}$, dependiendo de que símbolo X_a o X_b se añade en y_1.

Cuando se ha producido un número finito de estos ciclos, y no está presente una ocurrencia de X en las cadenas, éstas son recibidas por y_2, D se reemplaza por c en todas, y permanecen en el nodo para siempre. El nodo y_2 recoge todas las cadenas de L y cualquier cadena que llega a este nodo

pertenece a L. □

Resolución de problemas con HNEP

La utilización de HNEPs en la resolución de problemas se puede realizar como sigue. La computación en el HNEP asociado debe ser finita para toda instancia del problema. Esto indica que no existe un procesador especializado en inserciones. Si el problema planteado es un problema de decisión, obtendremos al final de la computación en el nodo de salida todas las soluciones del problema codificado con cadenas, si existe solución, si no el nodo nunca contendrá ninguna palabra. Si el problema requiere un conjunto finito de cadenas, este conjunto de cadenas se obtendrá en el nodo de salida al final de la computación. Puede ocurrir en otros casos, que el resultado se obtenga por métodos especiales que se indicarán para cada problema.

Un NEP homogéneo completo de tamaño $7m + 2$ *resuelve en* $O(m + n)$ *tiempo una* (n, m)*-instancia del problema de los 3 colores con* n *vértices y* m *aristas[13]. Siguiendo la descripción del formato para tres problemas NP-completos que se presentaron en [97] se presentará una solución algorítmica al problema común. Estos problemas son:*

1. *El problema de cobertura de vértices.*

   Dado un grafo no dirigido encontrar la cardinalidad de un conjunto mínimo de vértices tal que cada arista tenga como poco uno de sus extremos en este conjunto.

2. *El máximo conjunto independiente.*

   Dado un grafo no dirigido $G = (X, E)$*, donde* X *es el conjunto finito de vértices y* E *es el conjunto de aristas dadas como una familia de conjuntos de dos vértices, encontrar la cardinalidad del subconjunto máximo (con respecto a la inclusión) de* X *que no contiene ambos vértices conectados por cualquier arista en* E*.*

3. *El problema de satisfactibilidad (SAT).*

Dado un conjunto P de variables booleanas y un conjunto finito U de cláusulas sobre P, ¿existe una asignación de variables de P que satisfacen todas las cláusulas de U?

Para una formulación más detallada y discusión sobre sus soluciones, se recomienda [29].

Dichos problemas se pueden ver como casos especiales del denominado problema común algorítmico (CAP) en [97]: sea S un conjunto finito y F una familia no vacía de subconjuntos de S. Encontrar la cardinalidad de un subconjunto máximo de S que no incluye ningún conjunto perteneciente a F. Los conjuntos en F se denominan conjuntos prohibidos. Se dice que (F, S) es una $(card(S), card(F))$-instancia de CAP.

Estos tres problemas mencionados se pueden obtener como casos especiales de los CAP. El Problema de cobertura de vértices se obtiene con $S = X$ y F contiene todos los conjuntos $s(x) = \{x\} \cup \{y \in X \mid \{x, y\} \in E\}$. La cardinalidad que se busca es la diferencia entre la cardinalidad de S y la solución del CAP. El tercer problema se obtiene con $S = P \cup P'$, donde $P' = \{p' \mid p \in P\}$, y $F = \{F(C) \mid C \in U\}$, donde cada conjunto $F(C)$ asociado con la cláusula C se define por:

$$F(C) = \{p' \mid p \text{ aparece en } C\} \cup \{p \mid \neg p \text{ aparece en } C\}$$

Por otra parte para el problema de El máximo conjunto independiente, se toma $S = X$ y $F = E$.

Indicar que dada una instancia del problema de satisfactibilidad tiene una solución si y sólo si la solución de la instancia construida del CAP tiene exactamente la cardinalidad de P.

1.4.5. Redes de Procesadores Evolutivos con *Splicing Rules (NEPPS)*

Dado un NEPPS –NEP with Permiting context and Splicing rules – consideraremos un alfabeto V y dos símbolos especiales $\#, \$$ que no pertenecen a V. Una regla de mezcla (splicing rule) sobre V está formada por una cadena con la siguiente estructura:

$$r = u_1\#u_2\$u_3\#u_4, \text{ donde } u_1, u_2, u_3, u_4 \in V^*$$

La aplicación de una regla de splicing rule $r = u_1\#u_2\$u_3\#u_4$ sobre las cadenas $x, y, z \in V^*$ se denota por:

$$(x, y) \vdash_r z \text{ si y sólo si: } x = x_1u_1u_2x_2, y = y_1u_3u_4y_2, z = x_1u_1u_4y_2$$

para algún, $x_1, x_2, y_1, y_2 \in V^*$

La aplicación de una regla de splicing rule $r = u_1\#u_2\$u_3\#u_4$ sobre las cadenas $x, y, z, w \in V^*$ se denota por:

$$(x, y) \models_r (z, w) \text{ si y sólo si:}$$

$$x = x_1u_1u_2x_2, y = y_1u_3u_4y_2, z = x_1u_1u_4y_2, w = y_1u_3u_2x_2$$

para algún, $x_1, x_2, y_1, y_2 \in V^*$

En un conjunto de reglas de splicing rules R, el radio de R es:

$$rad(R) = max\{|x|, x = u_i, 1 \leq i \leq 4, \text{ para algún } u_1\#u_2\$u_3\#u_4 \in R\}$$

Una splicing rule con contexto permisivo (sobre V) es una tripleta de la forma:

$$p = (r; C_1, C_2), \text{ con } r = u_1\#u_2\$u_3\#u_4$$

siendo r una splicing rule sobre V y C_1, C_2 subconjuntos finitos de V^*, sólo se consideran sistemas con componentes finitos.

Se define $(x, y) \models_p (z, w)$ para $x, y, z, w \in V^*$ y $p = (r; C_1, C_2)$,si y sólo si

$(x, y) \models_r (z, w)$, *con todo elemento de* C_1 *apareciendo como una subcadena en* x *y todo elemento de* C_2 *figurando como una subcadena de* y*; cuando* $C_1 = \emptyset$ *o* $C_2 = \emptyset$ *entonces no se aplica ninguna condición en* x*, repectivamente en* y*.*

Sean P *y* F *dos subconjuntos disjuntos sobre un alfabeto* V *y una palabra en* V*, se definen los predicados* $\varphi^{(1)}$ *y* $\varphi^{(2)}$ *como:*

$$\varphi^{(1)}(w; P, F) \equiv P \subseteq alph(w) \wedge F \cap alph(w) = \emptyset$$

y

$$\varphi^{(2)}(w; P, F) \equiv alph(w) \cap P \neq \emptyset \wedge F \cap alph(w) = \emptyset$$

Las condiciones de contexto aleatorio definidas por los dos conjuntos P *(contexto permisivo) y* F *(contexto prohibitivo) delimitan la construcción de estos predicados. El contexto permisivo definido aquí es diferente que el contexto permisivo que emplea cada splicing rule. Para cada lenguaje* $L \subseteq V^*$ *y* $\beta \in \{(1), (2)\}$ *se define:*

$$\varphi^{\beta}(L, P, F) = \{w \in L | \varphi^{\beta}(w; P, F)\}$$

Una red de procesadores evolutivos con splicing rules y contextos Permisivos (NEPPS) de tamaño n *es un tupla:*

$$\Gamma = (V, N_1, N_2, \cdots, N_n, G)$$

donde V *es un alfabeto y para cada* $1 \leq i \leq n$*,*

$$N_i = (M_i, A_i, PI_i, FI_i, PO_i, FO_i, \beta_i)$$

es el i*-ésimo nodo evolutivo del procesador de la red y* G *es el llamado grafo subyacente de la red. Los parámetros de cada procesador son:*

- M_i *es el conjunto de splicing rules con contexto permisivo en el* i*-ésimo*

procesador.

- A_i *es el conjunto finito de palabras sobre* V*, presente inicialmente en el* i*-ésimo procesador.*
- $PI_i, FI_i \subseteq V$ *son los contextos permisivos y prohibitivos de entrada del procesador.*
- $PO_i, FO_i \subseteq V$ *son los contextos permisivos y prohibitivos de salida del procesador.*
- $\beta_i \in \{(1), (2)\}$ *define el tipo de los filtros de entrada/salida de un nodo. Más concretamente, para cada nodo,* $x \in G$*, se definen los siguientes filtros:*

  El filtro de entrada es:

  $$\rho_x(.) = \varphi^{\beta(x)}(.; PI_x, FI_x)$$

  y el filtro de salida es:

  $$\tau_x(.) = \varphi^{\beta(x)}(.; PO_x, FO_x)$$

  $\varphi_x(w)$ *(respectivamente* $\tau_x(w)$*) indica si la palabra* w *es capaz de atravesar o no el filtro de entrada (resp. salida) de* x*.* $\varphi_x(L)$ *(resp.* $\tau_x(L)$*) es el conjunto de palabras de* L *que pueden atravesar el filtro de entrada (resp. salida) de* x*.*

La configuración de este tipo de NEPs es una n*-tupla* $C = (L_1, L_2, \cdots, L_n)$ *con* $L_i \subseteq V^*$ *para todo* $1 \leq i \leq n$*.*
Una configuración representa los conjuntos de cadenas que están presentes en los nodos en un momento dado (cada cadena aparece un número de veces arbitrariamente grande). La configuración incial de la red es $C_0 = (A_1, A_2, \cdots, A_n)$*. Una configuración puede cambiar por un paso de evolución o de comunicación. Cuando cambia por un paso de evolución, cada componente de* L_i *de la configuración se cambia de acuerdo a las splicing rules asociadas al nodo* i*.*

La configuración $C_1 = (L_1, L_2, \cdots, L_n)$ *se transforma en la configuración* $C_2 = (L'_1, L'_2, \cdots, L'_n)$ *debido a un paso de evolución, y denotaremos formalmente por* $C_1 \Rightarrow C_2$*, si* L'_i *es el conjunto de cadenas resultado de la aplicación de las reglas de splicing rules del nodo* N_i *a las cadenas en* N_i*. Debido a la existencia de un número de copias de cadenas muy grande en cada nodo, obtendremos un gran número de copias de cadenas en cada uno de los nodos después de un paso de evolución al aplicar todas las posibles reglas de splicing rules para ese nodo. Diremos que, si* L_i *es vacío para algún* $1 \leq i \leq n$*, entonces* L'_i *también es vacío.*

Si la configuración cambia después de un paso de comunicación, cada uno de los procesadores N_i *enviará todas las copias de las cadenas que atraviesan su filtro de salida a cada nodo conectado con* N_i *y recibirá todas las copias de las cadenas enviadas por cualquier procesador conectado a* N_i *que atraviesen su filtro de entrada. Diremos que una configuración* C' *es obtenida en un paso de comunicación a partir de la configuración* C*, y denotaremos como* $C \vdash C'$*, si:*

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E} (\tau_y(C(y)) \cap \rho_x(C(y))) \text{ para todo } x \in G$$

Siendo E *el conjunto de vértices del grafo subyacente de la red.*
Una computación en Γ*, siendo* Γ *un NEPPS, es la secuencia de configuraciones* $C_0, C_1, \cdots$*, donde* C_0 *es la configuración inicial de* Γ *y* $C_{2i} \Rightarrow C_{2i+1}$ *y* $C_{2i+1} \vdash C_{2i+2}$ *para todo* $i \geq 0$*. Siendo la computación finita si la secuencia es finita.*
El resultado de una computación es un lenguaje que se obtiene en un nodo especial denomimado nodo de salida (o maestro) de la red.
Si el nodo k *es el nodo Maestro de la red , y si* $C_0, C_1, \cdots$ *es una computación, entonces todas las cadenas que existen en el nodo* k *en cualquier instante* t *en la* k*-ésima componente de* C_t *pertenece al lenguaje generado por la red. Este lenguaje es* $L_k(\Gamma)$*.*

Definiremos la complejidad sobre un conjunto finito de cadenas Z *en tiem-*

po de la computación como el mínimo número de pasos t en una computación $C_0, \cdots, C_t, \cdots$ *tal que Z sea un subconjunto de la k-ésima componente de* C_t.

Completitud Computacional

A continuación demostraremos que un NEPPS de dos nodos es completo computacionalmente, independientemente de la estructura del grafo subyacente.

Lema 1.4.10. $RE \subseteq EH_2(FIN, pFIN)$ *[80]*

Donde $EH_2([n], p[m]), n, m \geq 1$ *denota la familia de todos los lenguajes* $L(\gamma)$ *generados por los sistemas H extendidos con contextos permisivos,*$\gamma = (V, T, A, R)$ *con* $card(A) \leq n$ *y* $rad(R) \leq m$*, donde* $rad(R)$ *es el radio máximo de las splicing rules r en tripletas* $(r; C_1, C_2)$*. Cuando no hay restricción en el número de axiomas o en el radio máximo (excepto que estos números sean finitos), se puede reemplazar* $[n]$ *o* $[m]$ *por* FIN.

De donde se puede inferir el siguiente teorema.

Teorema 1.4.11. *Cualquier lenguaje enumerable recursivo se puede generar por un NEPPS completo de tamaño 2 donde las splicing rules son regulares.*

Demostración. Considerese una gramática *Chomsky* de tipo 0, $G = (N, T, S, P)$. Sea $U = N \cup T \cup \{B\}$, donde B es un símbolo nuevo. Se contruye el siguiente NEPPS de tamaño 2 con un grafo subyacente totalmente conectado:

$$\Gamma = (V, N_1, N_2, K_2)$$

donde:

$$V = N \cup T \cup \{B, X, X', Y, Z, Z', Z''\} \cup \{Y_\alpha | \alpha \in U\}$$

K_2 representa un grafo totalmente conectado de tamaño dos. Los parámetros para los diferentes nodos son:

- $N_1 = (M_1, \{XBSY, ZY, XZ, Z', Z'', ZY\} \cup \{Z_vY | u \to v \in P\} \cup \{ZY_\alpha, X'\alpha Z | \alpha \in U\}, N, T, T, N \cup \{B, X, X', Z, Z', Z'', Y, Y_\alpha\}, (2))$ y M_1 contiene reglas de la forma:

Simulate: 1.	$(\#uY\$Z\#vY; \{X\}, \emptyset)$	para $u \to v \in P$
Rotate: 2.	$(\#\alpha Y\$Z\#Y_\alpha; \{X\}, \emptyset)$	para $\alpha \in U$
Rotate: 3.	$(X\#\$X'\alpha\#Z; \{Y_\alpha\}, \emptyset)$	para $\alpha \in U$
Rotate: 4.	$(\#Y_\alpha\$Z\#Y; \{X'\}, \emptyset)$	para $\alpha \in U$
Rotate: 5.	$(X'\#\$X\#Z; \{Y\}, \emptyset)$	
Terminate: 6.	$(XB\#\$\#Z'; \{Y\}, \emptyset)$	
Terminate: 6.	$(\#Y\$Z''\#; \emptyset, \emptyset)$	

- $N_2 = (\emptyset, \emptyset, T, N \cup \{B, X, X', Z, Z', Z'', Y, Y_\alpha\}, N, \emptyset, (2))$

Este sistema se comporta exactamente igual que el del lema 1.4.10 descrito por [80]. Una vez que un terminal se genera en el nodo N_1, se pasa al nodo N_2 empleando los filtros. Ninguna otra cadena puede dejar el nodo N_1. Así se obtiene que $L(G) = L(\Gamma)$.

□

Parte II

Redes de Procesadores Evolutivos Masivamente Paralelos (MPNEP)

Capítulo 2

Redes de Procesadores Evolutivos Masivamente Paralelos (MPNEP)

2.1. Arquitectura de los MPNEP

Los modelos conexionistas consisten en varios procesadores que se localizan en los nodos de un grafo virtual y estos procesadores son capaces de realizar operaciones internas de acuerdo a determinadas reglas predefinidas. La información fluye a través de las conexiones para obtener una solución a un problema dado. Todos los procesadores trabajan en paralelo y tan sólo son capaces de operar de una manera sencilla [55].

Sea V un alfabeto sobre un conjunto de símbolos. Una cadena x de longitud m que pertenece al alfabeto V es una secuencia de símbolos $a_1 a_2 \cdots a_m$ donde el símbolo $a_i \in V$ para todo $1 \leq i \leq m$. El conjunto de todas las cadenas que pertenecen a V se denota por V^* y la cadena vacía se representa por ϵ.

Una Red de procesadores evolutivos [12, 13] de tamaño n (NEP) es una estructura $\Sigma = \{V, G, N_0, N_1, \cdots, N_n\}$, donde V es un alfabeto y los procesadores N_i están conectados en el grafo G.

Un procesador se define por $N_i = \{M_i, A_i, PI_i, PO_i\}$*, siendo* N_i *el i-ésimo procesador evolutivo de la red. Los parámetros de cada procesador son:*

- M_i *es un conjunto finito de reglas de evolución de una de las siguientes maneras (reglas básicas):*

  - $a \to b$, $a, b \in V$ *(reglas de sustitución),*
  - $a \to \epsilon$, $a \in V$ *(reglas de borrado),*
  - $\epsilon \to a$, $a \in V$ *(reglas de inserción).*

  Más claramente, el conjunto de reglas de evolución de un procesador está formado por reglas de sustitución, borrado o inserción. La información de contexto se puede incorporar a M_i *de la siguiente manera (reglas de contexto direccional):*

  - $ac \to bc$, $a, b \in V, c \in V^*$ *(reglas de sustitución por la izquierda),*
  - $ca \to cb$, $a, b \in V, c \in V^*$ *(reglas de sustitución por la derecha),*
  - $ab \to a$, $b \in V, a \in V^*$ *(reglas de borrado por la derecha),*
  - $ba \to a$, $b \in V, a \in V^*$ *(reglas de borrado por la izquierda),*
  - $a \to ab$, $b \in V, a \in V^*$ *(reglas de inserción por la derecha),*
  - $a \to ba$, $b \in V, a \in V^*$ *(reglas de inserción por la izquierda).*

  O incluso un contexto no direccional más general se puede expresar en M_i *(reglas de contexto):*

  - $dac \to dbc$, $a, b \in V, c, d \in V^*$
  - $abc \to ac$, $b \in V, a, c \in V^*$
  - $ac \to abc$, $b \in V, a, c \in V^*$

- A_i *es un conjunto finito de cadenas sobre* V*. El conjunto* A_i *es el conjunto de cadenas iniciales en el nodo* i*. Se considera que cada cadena que aparece en un nodo, lo es con un número arbitrario grande de copias, por lo que se indentifican los multiconjuntos por su soporte.*

- *PI_i y PO_i son subconjuntos de V que representan los filtros de entrada y salida respectivamente. Estos filtros se definen por condiciones de pertenencia, esto es, una cadena $w \in V^*$ puede atravesar el filtro de entrada(salida) si $\forall x \in PI_i, w = axb$ donde $a, b \in V^*$ ($\forall x \in PO_i, w = axb$ donde $a, b \in V^*$).*

  Se denota por $\rho_i(w) = true$, si w puede atravesar el filtro de entrada del nodo i y $\rho_i(w) = false$, en otro caso. Se denota $\tau_i(w) = true$, si w puede atravesar el filtro de salida del nodo i y $\tau_i(w) = false$, en otro caso.

La configuración de un NEP es una n-tupla $C = (L_0, L_1, \cdots, L_n)$, con $L_i \subseteq V^*$ para todo $0 \leq i \leq 6$. Una configuración representa el conjunto de todas las cadenas que están presentes en un nodo en un momento dado; claramente la configuración inicial de la red es $C_0 = (A_1, A_2, \cdots, A_n)$.

2.2. Dinámica de los MPNEP

Una configuración puede cambiar o bien por un paso de evolución o bien por un paso de comunicación. Los pasos de computación se pueden definir de una manera controlada, esto es, primero una fase de evolución y posteriormente una de comunicación; o de manera paralela, esto es, evolución y comunicación ocurren al mismo tiempo.

Cuando la red cambia por un paso de evolución, cada componente L_i de la configuración se cambia de acuerdo a las reglas de evolución asociadas con ese nodo i. Formalmente, se dice que una configuración $C_1 = (L_1, L_2, \cdots ., L_n)$ evoluciona hacia una configuración $C_2 = (L'_1, L'_2, \cdots, L'_n)$ por un paso de evolución, escrito como:

$$C_1 \Rightarrow C_2$$

si L'_i es el conjunto de todas las cadenas obtenidas aplicando las reglas de evolución de R_i a las cadenas pertenecientes a L_i como sigue:

- *Si la misma regla de sustitución puede reemplazar diferentes ocurrencias del mismo símbolo en una cadena, todas las ocurrencias se reemplazan pero empleando diferentes copias de esa cadena. El resultado es el multiconjunto en el cual cada cadena que se puede obtener aparece un número grande de veces.*

- *Las reglas de inserción y borrado se aplican sólo al final de la cadena. Así, una regla de borrado* $a \rightarrow \epsilon$ *se puede aplicar sólo sobre una cadena que finaliza con el simbolo* a*, esto es* wa *produce la cadena* w*, y una regla de inserción* $\epsilon \rightarrow a$ *aplicada a la cadena* x *consiste en añadir el símbolo* a *al final de* x*, obteniendo* xa*. Si se emplean reglas con contexto, esto es* $ab \rightarrow a$ *o* $abc \rightarrow ac$ *entonces el punto de borrado se define por la información de contexto* b *o* ac*.*

- *Si más de una regla, de cualquier tipo, se puede aplicar a una cadena, todas ellas emplearán diferentes copias de esa cadena.*

Ya que existe un gran número de copias de cada cadena en los nodos, después de un paso de evolución cada nodo obtendrá un número de copias grande de las cadenas que se pueden obtener aplicando las reglas del conjunto de reglas de evolución asociadas a los nodos. Por definición, si L_i *es vacío para algún* $0 \leq i \leq 6$*, entonces* L'_i *tambien es vacío.*

Cuando se produce un cambio por un paso de comunicación, cada nodo envía todas las copias de las cadenas que posee y que son capaces de atravesar su filtro de salida hacia los otros nodos y recibe todas las copias de las cadenas enviadas hacia él por cualquier nodo siempre y cuando puedan atravesar su filtro de entrada [55].

Formalmente, se dice que una configuración $C_1 = (L_1, L_2, \cdots, L_n)$ *cambia hacia la configuración* $C_2 = (L'_1, L'_2, \cdots, L'_n)$ *por un paso de comunicación, denotado por:*

$$C_1 \vdash C_2$$

si para todo $0 \leq i \leq n$*,*

$$L_i' = L_i \setminus \{w \in L_i | \tau_i(w) = true\} \cup \bigcup_{j=0, j \neq i}^{n} \{x \in L_j | \tau_j(x) = true \wedge \rho_i(x) = true\}$$

2.2.1. Computación paralela

Una computación paralela entre dos configuraciones C_1 y C_2, representada por $C_1 \models C_2$, consiste en la aplicación paralela de los pasos de evolución y comunicación, esto es:

$$C_1 \models C_2 = (C_1 \vdash C_2) || (C_1 \Rightarrow C_2)$$

Sea $\Gamma = (V, N_1, N_2, \cdots, N_n)$ un NEP. Una computación paralela en Γ es una secuencia de configuraciones $C_0, C_1, C_2, \cdots$, donde C_0 es la configuración inicial y $C_i \models C_{i+1}$ para todo $i \geq 0$. $MPNEP$ es un NEP_p (con computación paralela).

Si la secuencia es finita, se tiene una computación finita. El resultado de cualquier computación finita se recibe en un nodo designado como nodo de salida. Si se considera el nodo de salida de la red como el nodo N_0, y si $C_0, C_1, \cdots, C_t$ es una computación, entonces el conjunto de cadenas existentes en el nodo N_0 en el paso final –la componente 0-esima de C_t–, es el resultado de esta computación. La complejidad en tiempo de la computación es el número de pasos, esto es t.

2.2.2. Computación controlada

Sea $\Gamma = (V, N_1, N_2, \cdots, N_n)$ un NEP_c (c identifica computación controlada). Una computación controlada sobre Γ es una secuencia de configuraciones $C_0, C_1, C_2, \cdots$, donde C_0 es la configuración inicial y $C_{2i} \Rightarrow C_{2i+1}$ y $C_{2i+1} \vdash C_{2i+2}$ para todo $i \geq 0$.

Teorema 2.2.1. *Los problemas que se resuelven empleando un NEP_{cX} se*

pueden resolver empleando un MPNEP = NEP_{pX}, donde $X = \{b, d, c\}$ identifica el tipo de reglas (básicas, con contexto direccional, con contexto).

Demostración. Dado un procesador $N_i = \{A_i, R_i, PI_i, PO_i\}$ perteneciente a un NEP_{cX} es posible transformarlo en un procesador $N'_i = \{A_i, R'_i, PI_i, PO'_i\}$ que se comporta de la misma manera que el anterior pero en un $MPNEP = NEP_{pX}$ de la siguiente forma:

- Dada una regla $r_{ik} \in R_i$ con la notación $A \to B$, con $1 \leq k \leq p$, cada regla $r'_{ik} \in R'_i$ tiene la forma $A \to BX_{ik}$
- Dado el filtro de salida PO_i, $PO'_i = PO_i \bigcup_{k=1}^{p} X_{ip}$

Con estos conjuntos R'_i y PO'_i la computación paralela de un MPNEP se comporta de la misma manera que la computación controlada de un NEP ya que $\tau(w) = false$ hasta que se apliquen todas las reglas.

□

Teorema 2.2.2. *Los problemas que se resuelven empleando un NEP_{Xb} se pueden resolver empleando un NEP_{Xd} y los problemas resueltos empleando un NEP_{Xd} se pueden resolver con un NEP_{Xc}, donde $X = \{p, c\}$.*

Demostración. Es evidente que $abc \to adc$ donde $a, (c = \epsilon) \in V^*$ es equivalente a $ab \to ad$ y $ab \to ad$ donde $(a = \epsilon) \in V^*$ es equivalente a $b \to d$. □

Muchas variantes de NEPs se pueden definir dependiendo de las reglas y su dinámica, como se puede ver a continuación:

- *Computación controlada:*

Reglas básicas	*Reglas con contexto direccional*	*Reglas con contexto*
NEP_{cb}	NEP_{cd}	NEP_{cc}

- *Computación paralela:*

Reglas básicas	*Reglas con contexto direccional*	*Reglas con contexto*
NEP_{pb}	NEP_{pd}	NEP_{pc}

Es evidente que:

- $NEP_{cX} \subseteq NEP_{pX}$
- $NEP_{Xb} \subseteq NEP_{Xd} \subseteq NEP_{Xc}$

Por tanto, se puede establecer una relación de prioridad entre las diferentes clases de NEPs del siguiente modo:

- $NEP_{cb} \subseteq NEP_{cd} \subseteq NEP_{cc}$
- $NEP_{pb} \subseteq NEP_{pd} \subseteq NEP_{pc}$
- $NEP_{cb} \subseteq NEP_{pb}$
- $NEP_{cd} \subseteq NEP_{pd}$
- $NEP_{cc} \subseteq NEP_{pc}$

Pero no existe ningún indicio que determine:

- $NEP_{cc} \subseteq NEP_{pd}$ *or* $NEP_{cc} \nsubseteq NEP_{pd}$
- $NEP_{cc} \subseteq NEP_{pb}$ *or* $NEP_{cc} \nsubseteq NEP_{pb}$
- $NEP_{cd} \subseteq NEP_{pb}$ *or* $NEP_{cd} \nsubseteq NEP_{pb}$

2.3. Solución al problema de los 3 colores

A pesar de su simplicidad, las redes de procesadores evolutivos se pueden emplear para resolver problemas NP-completos, por ejemplo, el problema de los 3-colores, en tiempo lineal y con recursos lineales (nodos, símbolos, reglas).

Teorema 2.3.1. *El problema de los 3-colores se puede resolver en tiempo $O(m+n)$ empleando un NEP simple completo de tamaño $7m+2n$, donde n es el número de vértices y m es el número de aristas en el grafo de entrada. [12, 13]*

La idea consiste en construir un NEP donde los primeros $2n$ pasos, n son de Evolución y n son de comunicación pero en estos últimos no se comunicará nada hasta que no aparezca ninguna letra de T en las cadenas permanecen en N_0 . Cuando este proceso termina, las cadenas obtenidas codifican todas las posibles maneras de colorear los vértices, cumpliendo o no los requisitos del problema. Ahora, para cada arista e_t, el NEP mantiene solamente aquellas cadenas que codifican una condición de satisfactibilidad para los dos vértices de e_t.

Es bastante interesante que el grafo subyacente del NEP anteriormente mencionado no depende del número de nodos de la instancia del problema. En otras palabras, la misma estructura se puede emplear para resolver cualquier instancia del problema de los 3-colores siempre y cuando tengan el mismo número de aristas sin importar el número de nodos.

A continuación, mostraremos como se puede implementar un un NEP masivamente paralelo basado en la demostración del teorema 2.3.1, véase [12].

Teorema 2.3.2. *El problema de los 3-colores se puede resolver en tiempo $O(m+n)$ empleando un NEP masivamente paralelo de tamaño $4m+1n$, donde n es el número de vértices y m es el número de aristas del grafo de entrada.*

Demostración. Sea $G = (\{1, 2, \cdots, n\}, \{e_1, e_2, \cdots, e_m\})$ un grafo con $e_t =$

$\{k_t, l_t\}, 1 \leq k_t \leq l_t \leq n, 1 \leq t \leq m$. Se considera el alfabeto $U = V \cup V' \cup T \cup A$, donde $V = \{b, r, g\}$, $T = \{a_1, a_2, \cdots, a_n\}$, y $A = \{\hat{A}_1, \hat{A}_2, \cdots, \hat{A}_n\}$.

Se construyen los siguientes procesadores de un NEP masivamente paralelo.

- Un procesador generador:

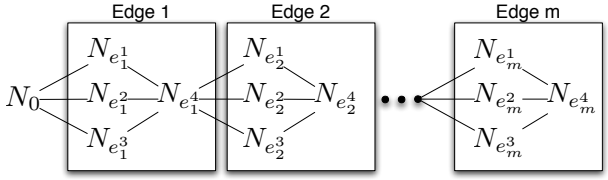
$$\begin{aligned} N_0 \;=\; & \{\{a_1 a_2 \cdots a_n\}, \{a_i \to b\hat{A}_i, a_i \to r\hat{A}_i, a_i \to g\hat{A}_i | 1 \leq i \leq n\}, \\ & \emptyset, \{\hat{A}_i | 1 \leq i \leq n\}\} \end{aligned}$$

  Este procesador genera todas las posibles combinaciones de colores, soluciones o no, al problema. Y envía estas cadenas a los siguientes procesadores.

- Para cada arista del grafo $e_t = \{k_t, l_t\}$, se tienen 4 procesadores de filtrado (donde $i = \{k_t, l_t\}$):

$$\begin{aligned} N_{e_t^1} &= \{\emptyset, \{g\hat{A}_i \to g' a_i, r\hat{A}_i \to r' a_i\}, \{\hat{A}_i\}, \{g', r'\}\} \\ N_{e_t^2} &= \{\emptyset, \{g\hat{A}_i \to g' a_i, b\hat{A}_i \to b' a_i\}, \{\hat{A}_i\}, \{g', b'\}\} \\ N_{e_t^3} &= \{\emptyset, \{b\hat{A}_i \to b' a_i, r\hat{A}_i \to r' a_i\}, \{\hat{A}_i\}, \{b', r'\}\} \\ N_{e_t^4} &= \{\emptyset, \{r' a_i \to r\hat{A}_i, g' a_i \to g\hat{A}_i, b' a_i \to b\hat{A}_i\}, \{a_i\}, \{\hat{A}_i\}\} \end{aligned}$$

Es claro que se puede construir un NEP paralelo con los procesadores anteriores de tal manera que N_0 genera todas las cadenas que codifican posibles combinaciones de color y posteriormente se aplican los procesadores $N_{e_t^1}, N_{e_t^2}, N_{e_t^3}, N_{e_t^4}$ para filtrar dichas cadenas en la arista e_t. La repetición de dicho proceso de filtrado para el resto de aristas provoca la obtención de una solución al problema.



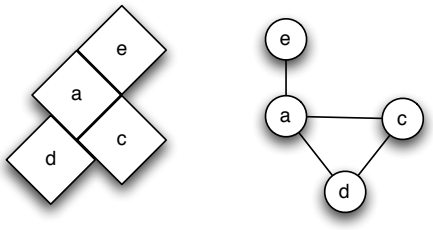


Figura 2.1: Instancia del problema de los 3-colores resuelta con un NEP masivamente paralelo.

Un NEP masivamente paralelo con la arquitectura anterior puede resolver el problema de los 3-colores de n nodos con m aristas. Como se puede apreciar, el tiempo total de la computación es $8m + n$. El número de reglas es $18m + 3n$. En conclusión, todos los parámetros de la red están acotados por $O(m + n)$.

□

2.3.1. Resultados de la simulación

Se ha desarrollado una herramienta basada en [55, 52, 56] para resolver el problema de los 3-colores. Este software emplea el modelo safe-thread de Java para simular las redes masivamente paralelas. Es evidente que la simulación no alcanza un tiempo lineal de computación ya que se ejecuta en una máquina secuencial, pero abre una plataforma para probar los resultados teóricos.

La figura 2.1 muestra el mapa de las nodos empleadas. La siguiente salida muestra el resultado de la computación del NEP anteriormente mencionado. Nótese que se han generado todas las soluciones válidas al problema.

```
Processor 16 : Objects: (12) [
   rAbCgDbE, gAbCrDbE, gAbCrDrE, rAgCbDgE, bAgCrDgE, rAgCbDbE,
   rAbCgDgE, bAgCrDrE, bArCgDrE, gArCbDrE, bArCgDgE, gArCbDbE
]
```

Donde $\{XY|X \in \{r(ed), g(reen), b(lue)\}, Y \in \{a, c, d, e\}\}$ *codifica el co-*

```
Processor 0 : Objects: (256) [ acde, rAcde, gAcde, bAcde, acrDe, acgDe, acbDe, arCde,
rAcrDe, rAcgDe, rAcbDe, gAcrDe, gAcgDe, gAcbDe, bAcrDe, bAcgDe, bAcbDe, arCrDe, arCgDe,
arCbDe, acdrE, acdgE, agCde, abCde, rArCde, rAgCde, rAbCde, gArCde, gAgCde, gAbCde, bArCde,
bAgCde, bAbCde, agCrDe, abCrDe, agCgDe, abCgDe, agCbDe, abCbDe, rArCrDe, rAgCrDe, rAbCrDe,
rArCgDe, rAgCgDe, rAbCgDe, rArCbDe, rAgCbDe, rAbCbDe, gArCrDe, gAgCrDe, gAbCrDe, gArCgDe,
gAgCgDe, gAbCgDe, gArCbDe, gAgCbDe, gAbCbDe, bArCrDe, bAgCrDe, bAbCrDe, bArCgDe, bAgCgDe,
bAbCgDe, bArCbDe, bAgCbDe, bAbCbDe, arCdrE, agCdrE, abCdrE, arCdgE, agCdgE, abCdgE, acdbE,
rAcdrE, rAcdgE, rAcdbE, gAcdrE, gAcdgE, gAcdbE, bAcdrE, bAcdgE, bAcdbE, acrDrE, acrDgE,
acrDbE, acgDrE, acgDgE, acgDbE, acbDrE, acbDgE, acbDbE, arCdbE, rAcrDrE, rAcrDgE, rAcrDbE,
rAcgDrE, rAcgDgE, rAcgDbE, rAcbDrE, rAcbDgE, rAcbDbE, agCdbE, abCdbE, rArCdrE, rAgCdrE,
rAbCdrE, rArCdgE, rAgCdgE, rAbCdgE, rArCdbE, rAgCdbE, rAbCdbE, gArCdrE, gAgCdrE, gAbCdrE,
gArCdgE, gAgCdgE, gAbCdgE, gArCdbE, gAgCdbE, gAbCdbE, bArCdrE, bAgCdrE, bAbCdrE, bArCdgE,
bAgCdgE, bAbCdgE, bArCdbE, bAgCdbE, bAbCdbE, arCrDrE, agCrDrE, abCrDrE, arCrDgE, agCrDgE,
abCrDgE, arCrDbE, agCrDbE, abCrDbE, arCgDrE, agCgDrE, abCgDrE, arCgDgE, agCgDgE, abCgDgE,
arCgDbE, agCgDbE, abCgDbE, arCbDrE, agCbDrE, abCbDrE, arCbDgE, agCbDgE, abCbDgE, arCbDbE,
agCbDbE, abCbDbE, rArCrDrE, rAgCrDrE, rAbCrDrE, rArCrDgE, rAgCrDgE, rAbCrDgE, rArCrDbE,
rAgCrDbE, rAbCrDbE, rArCgDrE, rAgCgDrE, rAbCgDrE, rArCgDgE, rAgCgDgE, rAbCgDgE, rArCgDbE,
rAgCgDbE, rAbCgDbE, gAcrDrE, gAcrDgE, gAcrDbE, gAcgDrE, gAcgDgE, gAcgDbE, gAcbDrE, gAcbDgE,
gAcbDbE, bAcrDrE, bAcrDgE, bAcrDbE, bAcgDrE, bAcgDgE, bAcgDbE, bAcbDrE, bAcbDgE, bAcbDbE,
rArCbDrE, rArCbDgE, rArCbDbE, rAgCbDrE, rAgCbDgE, rAgCbDbE, rAbCbDrE, rAbCbDgE, rAbCbDbE,
gArCrDrE, gArCrDgE, gArCrDbE, gAgCrDrE, gAgCrDgE, gArCgDrE, gArCbDrE, gAgCgDrE, gAgCbDrE,
gAbCrDrE, gAbCgDrE, gAbCbDrE, gArCgDgE, gArCbDgE, gAgCgDgE, gAgCbDgE, gAbCrDgE, gAbCgDgE,
gAbCbDgE, gArCgDbE, gArCbDbE, gAgCrDbE, gAgCgDbE, gAgCbDbE, gAbCrDbE, gAbCgDbE, gAbCbDbE,
bArCrDrE, bArCgDrE, bArCbDrE, bAgCrDrE, bAgCgDrE, bAgCbDrE, bAbCrDrE, bAbCgDrE, bAbCbDrE,
bArCrDgE, bArCgDgE, bArCbDgE, bAgCrDgE, bAgCgDgE, bAgCbDgE, bAbCrDgE, bAbCgDgE, bAbCbDgE,
bArCrDbE, bArCgDbE, bArCbDbE, bAgCrDbE, bAgCgDbE, bAgCbDbE, bAbCrDbE, bAbCgDbE, bAbCbDbE ]
```

Figura 2.2: Objetos en el procesador N_0 después de aplicar las reglas de evolución en $n = 4$ pasos.

lor de las nodos, esto es, X denota el color de la ciudad Y en el mapa, véase la figura 2.1. La figura 2.2 muestra los objetos en el procesador N_0 tras aplicar las reglas de evolución. Dicho procesador contiene 256 *objetos, este conjunto se ha obtenido en $n = 4$ pasos de evolución.*

Parte III

Redes de Procesadores Evolutivos con Filtros en las Conexiones(ANSPFC)

Capítulo 3

Redes de Procesadores Evolutivos con Filtros en las Conexiones

3.1. Red de procesadores con *splicing (ANSP)*

Un regla de splicing sobre un alfabeto V es una 4-tupla escrita en la forma $\sigma = [(x, y); (u, v)]$, donde x, y, u, v son palabras sobre V. Dada una regla splicing σ sobre V y un par de cadenas (w, z) sobre V se define la acción de σ sobre (w, z) como:

$$\sigma(w,z) = \{t | w = \alpha xy\beta, z = \psi uv\delta \text{ para algún } \alpha, \beta, \psi, \delta \\ y\ t = \alpha xv\delta \text{ o } t = \psi uy\beta\} \tag{3.1}$$

Esta operación sobre un par de cadenas se puede extender a un par de lenguajes A, B:

$$\sigma(A,B) = \bigcup_{w \in A, z \in B} \sigma(w,z) \tag{3.2}$$

Además, si M es un conjunto finito de reglas de splicing sobre V:

$$M(A,B) = \bigcup_{\sigma \in M} \sigma(A,B) \tag{3.3}$$

Sobre dos subconjuntos disjuntos P y F del alfabeto V y una palabra w de V, se definen los predicados:

$$\varphi^{(1)}(w;P,F) \equiv P \subseteq alph(w) \wedge F \cap alph(w) = \emptyset \tag{3.4}$$
$$\varphi^{(2)}(w;P,F) \equiv alph(w) \subseteq P \tag{3.5}$$
$$\varphi^{(3)}(w;P,F) \equiv P \subseteq alph(w) \wedge F \not\subseteq alph(w) \tag{3.6}$$
$$\varphi^{(4)}(w;P,F) \equiv alph(w) \subseteq P \wedge F \cap alph(w) = \emptyset \tag{3.7}$$

La construcción de estos predicados está basada en condiciones de contexto aleatorias definidas por los dos conjuntos P (contexto permitido) y F (contexto prohibido). Informalmente, la primera condición requiere que todos los símbolos permitidos aparezcan en w y que no aparezca ningún símbolo de los prohibidos; la segunda que todos los símbolos de w sean permitidos, mientras que las dos últimas son variantes más relajadas de las dos anteriores [56].

Para cualquier lenguaje $L \subseteq V^*$ y $\beta \in \{(1),(2),(3),(4)\}$ se define:

$$\varphi^{\beta}(L,P,F) = \{w \in L | \varphi^{\beta}(w;P,F)\} \tag{3.8}$$

Un procesador con splicing sobre V es una 6-tupla (S,A,PI,FI,PO,FO) donde:

- *S es un conjunto finito de reglas de splicing sobre V.*
- *A es un conjunto finito de palabras auxiliares sobre V. Estas palabras auxiliares se emplean por el nodo para realizar la operación de splicing.*

- *$PI, FI \subseteq V$ son los contextos de entrada permitidos/prohibidos del procesador, mientras que $PO, FO \subseteq V$ son los contextos de salida permitidos/prohibidos del procesador (con $PI \cap FI = \emptyset$ y $PO \cap FO = \emptyset$).*

El conjunto de procesadores sobre V se denota por SP_V.

Una red de procesadores evolutivos con splicing (ANSP) está formada por una 6-tupla $\Gamma = (V, U, G, \mathcal{N}, \alpha, x_I, x_O)$, donde:

- *V y U son los alfabetos de entrada y de la red, respectivamente, $V \subseteq U$.*
- *$G = (X_G, E_G)$ es un grafo no dirigido con el conjunto de vértices X_G y el conjunto de aristas E_G. G se denomina el grafo subyacente de la red.*
- *$\mathcal{N} : X_G \rightarrow SP_U$ es una función que asocia a cada nodo $x \in X_G$ el procesador de splicing $\mathcal{N}(x) = (S_x, A_x, PI_x, FI_x, PO_x, FO_x)$.*
- *$\alpha : X_G \rightarrow \{(1), (2), (3), (4)\}$ define el tipo de los filtros de entrada/salida de un nodo. Para cada nodo $x \in X_G$ se definen los siguientes filtros:*

  $$\textit{filtro de entrada: } \rho_x(.) = \varphi^{\beta(x)}(.; PI_x, FI_x)$$

  $$\textit{filtro de salida: } \tau_x(.) = \varphi^{\beta(x)}(.; PO_x, FO_x)$$

  Esto es, $\rho_x(w)$ (respectivamente τ_x) indica si una palabra w puede atravesar o no el filtro de entrada (salida) de x. $\rho_x(L)$ (respectivamente $\tau_x(L)$) es el conjunto de palabras de L que atraviesan el filtro de entrada (salida) de x.
- *$x_I, x_O \in X_G$ son los nodos de entrada y salida de Γ.*

Se dice que $card(X_G)$ es el tamaño de Γ. Si α es una función constante, entonces la red se dice que es homogénea. En la teoría de redes algunos tipos de grafos subyacentes son: anillo; estrella; malla; etc. Las redes de procesadores evolutivos pueden tener estos grafos subyacentes. Este trabajo se centra en $ANSP$ completos, esto es $ANSP$ con un grafo completo denotado por K_n, donde n es el número de vértices.

La configuración de un ANSP es una función $C : X_G \to 2^{V^*}$ *que asocia a cada nodo del grafo un conjunto de palabras. Una configuración se puede entender como los conjuntos de palabras que están presentes en los nodos de la red en un momento dado. Dada una palabra* $w \in V^*$*, la configuración inicial de* Γ *en* w *se define por* $C_0^{(w)}(x_I) = w$ *y* $C_0^{(w)} = \emptyset$ *para todo* $x \in X_G - \{x_I\}$*.*

Una configuración puede cambiar por un paso de splicing o por un paso de comunicación. Cuando cambia por un paso de splicing, cada componente de $C(x)$ *de la configuración* C *se cambia de acuerdo al conjunto de reglas de splicing* M_x *asociadas al nodo* x *y el conjunto* A_x*. Formalmente, se dice que la configuración* C' *se obtiene con un paso de splicing a partir de la configuración* C*, denotado por* $C \Rightarrow C'$*, si:*

$$C'(x) = S_x(A_x, C(x)) \text{ para todo } x \in X_G$$

Cuando cambia por un paso de comunicación, cada nodo $x \in X_G$ *envía una copia de cada palabra que posee, que es capaz de atravesar el filtro de salida de* x*, a todos los nodos conectados a* x *y recibe todas las cadenas enviadas por cualquier nodo conectado a* x *siempre y cuando puedan atravesar su filtro de entrada.*

Formalmente, se dice que una configuración C' *se obtiene en un paso de comunicación a partir de la configuración* C*, denotado por* $C \vdash C'$*, si:*

$$C'(x) = (C(x) - \tau_x(C(x))) \cup \bigcup_{\{x,y\} \in E_G} (\tau_y(C(y)) \cap \rho_x(C(y))) \text{ para todo } x \in X_G$$

Sea Γ *un ANSP, la computación de* Γ *sobre la palabra de entrada* $w \in V^*$ *es una secuencia de configuraciones* $C_0^{(w)}, C_1^{(w)}, C_2^{(w)}, \cdots,$*, donde* $C_0^{(w)}$ *es la configuración inicial de* Γ *sobre* w*,* $C_{2i}^{(w)} \Rightarrow C_{2i+1}^{(w)}$ *y* $C_{2i+1}^{(w)} \vdash C_{2i+2}^{(w)}$*, para todo* $i \geq 0$*. La configuración* $C_i^{(w)}$ *está univocamente determinada por* $C_{i-1}^{(w)}$*.*
Cada computación en un ANSP es determinista. Una computación finaliza (y se dice que es finita) si se cumple alguna de estas dos condiciones:

- *Existe una configuración en la cual el conjunto de cadenas existentes en el nodo x_O es no vacío. En este caso, la computación se dice que es una computación de aceptación.*
- *Existen dos configuraciones idénticas.*

El lenguaje aceptado por Γ *es:*

$$L_a(\Gamma) = \{w \in V^* | \text{ la computación de } \Gamma \text{ sobre } w \text{ es de aceptación } \}$$

Se dice que un ANSP decide el lenguaje $L \subseteq V^*$*, denotado por* $L(\Gamma) = L$ *si y sólo si* $L_a(\Gamma) = L$ *y la computación de* Γ *sobre cualquier* $x \in V^*$ *se detiene.*

De manera similar, se definen dos medidas de complejidad computacional empleando un ANSP como modelo de cómputo.
Para conseguir esto se considera un ANSP Γ *con el alfabeto de entrada* V *que se detiene en cualquier entrada.*
La complejidad en tiempo de la computación finita $C_0^{(x)}, C_1^{(x)}, C_2^{(x)}, \cdots, C_m^{(x)}$ *en* Γ *sobre* $x \in V^*$ *se denota por* $Time_\Gamma(x)$ *e igual a* m*. La complejidad en longitud de la computación se define por:*

$$Length_\Gamma(x) = max\{|w| : w \in C_i^{(x)}(z), 1 \leq i \leq m, z \in X_G\}$$

La complejidad en tiempo de Γ *es la función parcial* $\mathbf{N}$ *a* $\mathbf{N}$*:*

$$Time_\Gamma(x) = max\{Time_\Gamma(x) | x \in V^*, |x| = n\}$$

Y la complejidad en longitud de Γ *es la función parcial* $\mathbf{N}$ *a* $\mathbf{N}$*:*

$$Length_\Gamma(n) = max\{Length_\Gamma(x) | x \in V^*, |x| = n\}$$

Es fácil demostrar que para cualquier ANSP Γ *si* $Time_\Gamma$ *está acotado por un polinomio lineal, entonces* $Length_\Gamma$ *está tambien acotado.*

3.1.1. Resolución de problemas con $ANSP$

A continuación se discute brevemente e informalmente como un $ANSP$ se puede emplear para resolver problemas. Una posible correspondencia entre los problemas de decisión y los lenguajes se puede realizar a través de una función de codificación que transforma la instancia de un determinado problema de decisión en una cadena.
Se dice que un problema de decisión se resuelve en tiempo lineal por un $ANSP$ si se satisfacen las siguientes condiciones:

1. *La función de codificación se puede computar por una máquina de Turing determinista en tiempo lineal. Por tanto cada instancia del problema está linealmente relacionada con la palabra asociada.*

2. *Para cada instancia del problema se puede construir un $ANSP$ que decide en tiempo lineal la palabra que codifica la instancia dada. Esto quiere decir que la palabra es aceptada si y sólo si la solución a la instancia del problema es "SI". Esta construcción se denomina solución en tiempo lineal del problema considerado.*

Teorema 3.1.1. *El problema SAT se puede resolver en tiempo lineal con un $ANSP$. Además los recursos del $ANSP$ están acotados linealmente por el tamaño de la instancia del SAT.*

Demostración. Sea V el conjunto de variables, $V = \{x_1, x_2, \cdots, x_n\}$ y $\phi = (C_1) \wedge (C_2) \wedge \cdots (C_m)$ una fórmula booleana, donde la negación de la variable x_i se denota por $\bar{x}_i$. Cada fórmula se puede ver como una palabra sobre el alfabeto $V \cup \bar{V} \cup \{\wedge, \vee, (,)\}$, donde $\bar{V} = \{\bar{x} | x \in V\}$.
Se define el alfabeto:

$$W = \{[x_i = 1], [x_i = 0] | 1 \leq i \leq n\} \cup V \cup \bar{V} \cup \{\vee, \wedge, (,), \$, \#, \uparrow\}$$

Se considera el $ANSP$ $\Gamma = (V, W, K_{2n+2}, \mathcal{N}, \alpha, In, Out)$, donde K_{2n+2} es el grafo completo con $2n + 2$ nodos: In, Out, $(x_i \leftarrow 0)$, $(x_i \leftarrow 1)$ con

$1 \leq i \leq n$.
Cada nodo se define como:

- In:
  - $S_{In} = \{[(\$[x_n = b], \epsilon); (\$, ()]\ b \in \{0,1\}\} \cup \{[(\$[x_i = b], \epsilon); (\$, [x_{i+1} = c])|b, c \in \{0,1\}, 1 \leq i \leq n-1\}$
  - $A_{In} = \{\$[x_i = b]|b \in \{0,1\}, 1 \leq i \leq n\}$
  - $PI_{In} = \emptyset, FI_{In} = W, PO_{In} = \{[x_1 = 0, [x_1 = 1]\}, FO_{In} = \emptyset, \alpha(In) = (4)$
- $(x_1 \leftarrow 0), 1 \leq i \leq n$:
  - $S_{(x_i \leftarrow 0)}$ contiene todas las reglas de *splicing* de la forma $[(\uparrow, C'\beta\#); ([x_n = b], (C)\beta\#)]$, donde

    (i) $b \in \{0,1\}, C)\beta$ es un sufijo de ϕ, con $C \in (V \cup \bar{V} \cup \{\vee\})^+$

    (ii) C' $\begin{cases} \epsilon, \text{ si } C \text{ comienza con } \bar{x}_i \\ (\gamma), \text{ si } C = x_i \vee \gamma, \gamma \in (V \cup \bar{V})^+, \\ \uparrow, \text{ si } C = x_i \text{ o } C \text{ comienza con } x_j \text{ o } \bar{x}_j \text{ para algún } j \neq i \end{cases}$
  - $A_{(x_i \leftarrow 0)} = \{\uparrow C'\beta\#|C', \beta \text{ son los definidos anteriormente}\}$
  - $PI_{(x_i \leftarrow 0)} = \{[x_i = 0]\}, FI_{(x_i \leftarrow 0)} = \{\uparrow\}, PO_{(x_i \leftarrow 0)} = \emptyset, FO_{(x_i \leftarrow 0)} = \emptyset, \alpha((x_i \leftarrow 0)) = (1)$
- $(x_1 \leftarrow 1), 1 \leq i \leq n$:
  - $S_{(x_i \leftarrow 1)}$ contiene todas las reglas de *splicing* de la forma $[(\uparrow, C'\beta\#); ([x_n = b], (C)\beta\#)]$, donde

    (i) $b \in \{0,1\}, C)\beta$ es un sufijo de ϕ, con $C \in (V \cup \bar{V} \cup \{\vee\})^+$

    (ii) C' $\begin{cases} \epsilon, \text{ si } C \text{ comienza con } \bar{x}_i \\ (\gamma), \text{ si } C = x_i \vee \gamma, \gamma \in (V \cup \bar{V})^+, \\ \uparrow, \text{ si } C = x_i \text{ o } C \text{ comienza con } x_j \text{ o } \bar{x}_j \text{ para algún } j \neq i \end{cases}$
  - $A_{(x_i \leftarrow 1)} = \{\uparrow C'\beta\#|C', \beta \text{ son los definidos anteriormente}\}$
  - $PI_{(x_i \leftarrow 1)} = \{[x_i = 0]\}, FI_{(x_i \leftarrow 1)} = \{\uparrow\}, PO_{(x_i \leftarrow 1)} = \emptyset, FO_{(x_i \leftarrow 1)} = \emptyset, \alpha((x_i \leftarrow 1)) = (1)$

- Out:
  - $S_{Out} = A_{Out} = PI_{Out} = PO_{Out} = \emptyset$
  - $FI_{Out} = V \cup \bar{V} \cup \{(,)\vee, \wedge, \uparrow\}, FO_{Out} = W, \alpha(Out) = (1)$

El modo de funcionamiento de Γ sobre la palabra de entrada $w = \$\phi\#$. Donde se puede asumir que no existen cláusulas idénticas en ϕ. En la configuración inicial la palabra w está en el nodo de entrada In. Durante los $2n-1$ primeros pasos de computación, de los cuales n son de *splicing*, no se comunica ninguna palabra ya que no pueden abandonar el nodo In. Más formalmente, después de k pasos de *splicing* todas las palabras

$$\$[x_{n-k+1} = b_k] \cdots [x_n = b_1]\phi\#$$

con $b_j \in \{0,1\}, 1 \leq j \leq k$ están en In. Después de los primeros $2n-1$ pasos todas estas cadenas se comunican a los siguientes nodos. Todas estas cadenas tienen dos partes: un prefijo donde se asigna un 0 ó 1 a cada variable, denominado el prejifo-valor, y otra parte que consiste en otra cadena representando el sufijo de la fórmula de entrada, denominado sufijo fórmula. Todas las cadenas de esta forma se denominan cadenas correctas. Toda cadena que contiene $[x_k = b_k], k \in \{1, \cdots, n\}$ en su prefijo-valor puede entrar en el nodo $N(x_k \leftarrow 1)$ si $b_k = 1$, o en el nodo $N(x_k \leftarrow 0)$, en caso contrario.

Supongamos que la cadena entra en el nodo $N(x_k \leftarrow 1)$. Si la fórmula de entrada tiene la forma $(x_k \vee F) \wedge G$, entonces el sufijo fórmula de la cadena se reemplaza por G, que todavía representa el sufijo fórmula de la cadena. Si la fórmula de entrada es de la forma $(\bar{x}_k \vee F) \wedge G$, entonces el sufijo fórmula de la cadena se reemplaza por $(F) \vee G$, que sigue siendo el sufijo fórmula de la cadena. Finalmente, si el sufijo fórmula tiene la forma $(\bar{x}_k)G$, entonces se reemplaza por $\uparrow G$ que no es un sufijo fórmula, esto es una cadena incorrecta. Es fácil notar que todas las cadenas incorrectas se pierden tan pronto como abandonan el nodo donde se produjeron. La forma especial de las reglas de *splicing* requieren que la operación se lleve a cabo entre el prefijo valor y el sufijo fórmula de una cadena correcta. Por tal motivo, cualquier cadena que tenga un sufijo fórmula que comience por x_j o

$\bar{x}_j$ se transforma en una cadena incorrecta tan pronto como entra en el nodo $N(x_k \leftarrow b)$ con $k \neq j$. Ahora el proceso es iterativo, es claro que después de éstos pasos (*splicing*/comunicación), las cadenas correctas se transforman en cadenas correctas con una menor longitud en el sufijo fórmula o en cadenas incorrectas. Por tanto, el nodo *Out* contiene una palabra después de, como mucho, $2n + 2|\phi|$, donde $|\phi|$ denota la longitud de la fórmula ϕ, si y sólo si existe una asignación de n variables que satisfacen la fórmula ϕ dada. Si tal asignación no existe, entonces el $ANSP$ se detiene después de $2n + 2|\phi| + 1$ pasos con el nodo *Out* vacío.

También hay que matizar que los otros recursos de Γ están linealmente acotados: esto es trivial dado el tamaño de Γ y el número de símbolos siempre y cuando el número de palabras auxiliares y reglas de *splicing* en cada nodo esté linealmente acotado por la longitud de la fórmula de entrada. □

Cabe destacar que la estructura subyacente no cambia si el número de variables de la instancia permanece constante.
También se puede decir que la red es un programa: se escojen los filtros, las reglas de splicing, se adivina una asignación para las variables y se computa la fórmula reemplazando las variables por sus valores, uno a uno, de izquierda a derecha.

El problema HPP consiste en decidir si un grafo dirigido tiene un camino Hamiltoniaho. Un camino Hamiltoniano en un grafo dirigido es un camino que contiene todos los vertices, pero sólo una vez. Es conocido que el problema HPP es un problema NP-completo.

Teorema 3.1.2. *El problema HPP se puede resolver con un $ANSP$ en tiempo lineal. Además los recursos de la red están acotados linealmente por el número de nodos del grafo HPP.*

Demostración. Se considera el siguiente grafo dirigido $\gamma = (V, E)$, con $V = \{x_1, x_2, \cdots, x_n\}$ para el cual se está buscando un camino Hamiltoniano comenzando por x_1. Primeramente se define el alfabeto $U = V \cup \{\$, \#, \perp\}$ y el $ANSP$ homogéneo $\Gamma = (V, U, K_{n+1}, \mathcal{N}, \alpha, In, Out)$, donde K_{n+1} es el grafo completo con los nodos $In, y_2, y_3, \cdots, y_n, Out$, y el resto de parámetros son:

- In:
  - $S_{In} = \{[\$, x_1\#^{n-1}); (\perp, \#)]\}$
  - $A_{In} = \{\$x_1\#^{n-1}\}$
  - $PI_{In} = \emptyset, FI_{In} = V, PO_{In} = FO_{In} = \emptyset, \alpha(In) = (1)$
- $y_i, 2 \leq i \leq n$:
  - $S_{y_i} = \{[\$, x_i\#^{p-1}); (x_j, \#^p)] | (x_j, x_i) \in E, p \geq 1\} \cup$
    $\{[(\$, \$); (x_j, \#^p)] | (x_j, x_i) \notin E, p \geq 1\}$
  - $A_{y_i} = \{\$x_i\#^p | p \geq 0\} \cup \{\$\$\}$
  - $PI_{y_i} = \emptyset, FI_{y_i} = \{x_i, \$\}, PO_{y_i} = FO_{y_i} = \emptyset, \alpha(y_i) = (1)$
- Out:
  - $S_{Out} = A_{Out} = \emptyset, PI_{Out} = PO_{Out} = \emptyset$
  - $FI_{Out} = \{\$, \#\}, FO_{Out} = U, \alpha(Out) = (1)$

Supongamos que se comienza con la palabra $\perp \#^n$ en el nodo In. Después de un paso de *splicing*, dos cadenas abandonan el nodo In: $\$\#^n$ y $\perp x_1\#^{n-1}$. La primera se pierde mientras una copia de la segunda entra en cada nodo y_i, $2 \leq i \leq n$. Es fácil demostrar que cualquier palabra que contenga el símbolo \$ resultante de una operación de *splicing* en cualquier nodo se pierde en la comunicación. Sigamos la evolución de la cadena $\perp x_1\#^{n-1}$ que entra en el nodo y_i para algún i. El siguiente paso de *splicing* produce una cadena que continúa el proceso computacional, $\perp x_1x_i\#^{n-2}$, si y sólo si $(x_1, x_i) \in E$. Las cadenas obtenidas de esta manera contienen, entre los símbolos $\perp$ y la primera ocurrencia de #, caminos en G. Nótese que el número de ocurrencias de # en el sufijo de estas cadenas almacena el número de nodos que se necesitan para completar el camino Hamiltoniano. Además, una cadena que contiene el símbolo x_j, para algún j, no puede entrar de nuevo en el nodo y_j. Por tales motivos, después de $2n$ pasos el nodo Out contiene todos los caminos Hamiltonianos, si existen, o la computación se detiene después de como mucho $2n - 2$ pasos si el grafo no tiene camino Hamiltoniano. □

3.2. Red de procesadores con filtros en las conexiones (ANSPFC)

Una red de procesadores evolutivos con filtros en las conexiones, en adelante (ANSPFC) –Accepting Network of Splicing Proccessor with Filter Connections–, se define como una 9*-tupla:*

$$\Gamma = (V, U, <, >, G, \mathcal{N}, \alpha, x_I, x_O) \tag{3.9}$$

donde:

- V *es el alfabeto de entrada y* U *el alfabeto de la red,* $V \subseteq U$*, donde,* $<, > \in U \setminus V$ *serán símbolos definidos para un uso particular.*
- $G = (X_G, E_G)$ *es un grafo no dirigido sin bucles donde,* X_G *es el conjunto de nodos y* E_G *es el conjunto de aristas. Cada procesador splicing estará formado por un nodo* $x \in X_G$*, que posee un conjunto de reglas splicing* M_x *y un conjunto de axiomas* A_x*. Cada una de las arista será definida por un conjunto binario y* G *es el grafo subyacente de la red.*
- $\mathcal{N} : E_G \to 2^U \times 2^U$ *es la función que asocia con cada arista* $e \in E_G$ *los conjuntos disjuntos* $\mathcal{N}(e) = (P_e, F_e)$*.*
- $\alpha : E_G \to \{s, w\}$ *definirá el tipo de filtro utilizado en cada arista.*
- $x_I, x_O \in X_G$ *son los nodos de entrada y de salida de* Γ*, respectivamente.*

La $card(X_G)$ *es el tamaño de* Γ*. Nos centraremos en* $ANSPFC$ *completos, se entiende como* $ANSPFC$ *completo aquellos cuyo grafo subyacente es completo al que denotaremos como* K_n*, donde* n *es el número de nodos. Cabe reseñar que cualquier grafo subyacente sobre un* $ANSPFC$ *puede ser transformado en completo sin modificar sus características computacionales,*

añadiendo las aristas necesarias para completar el grafo subyacente asociando a cada una de estas nuevas aristas filtros con funcionalidad nula. Esta característica no es posible de implementar en los $ANSP$.

Se define configuración de un $ANSPFC$ como una función $C : X_G \rightarrow 2^{U^*}$ donde para cada nodo del grafo se le asocia un conjunto. Una configuración por tanto estará formada por los conjuntos de palabras que están presentes en los nodos en un instante T. Siendo $z \in V^*$ una palabra, diremos que la configuración inicial de Γ en z vendrá definida por $C_0^{(z)}(x_I) = \{< z >\}$ y $C_0^{(z)}(x) = \emptyset$ para todo $x \in X_G \setminus \{x_I\}$. Las palabras utilizadas como auxiliares no aparecen en ninguna configuración.

Un paso de splicing o un paso de comunicación pueden cambiar una configuración. Cuando una configuración cambia por un paso de splicing, cada componente de $C(x)$ de la configuración C se cambia de acuerdo al conjunto de reglas splicing M_x asociadas con el nodo x y el conjunto A_x. Formalmente, diremos que la configuración C' es obtenida por un paso de splicing a partir de la configuración C, denotado por $C \Rightarrow C'$, si y sólo si:

$$C'(x) = S_x(C(x) \cup A_x) \forall x \in X_G \tag{3.10}$$

Teniendo en cuenta que cada palabra presente en un nodo, así como cada palabra auxiliar, aparecerá un número ilimitado de veces, todas las posibles aplicaciones de las operaciones de splicing se puede presuponer que se efectúan en un único paso de splicing. Si el paso de splicing se define como $C \Rightarrow C'$, si y sólo si:

$$C'(x) = S_x(C(x), A_x) \forall x \in X_G \tag{3.11}$$

ésta característica definirá todos los procesadores de Γ como restrictivos.

Cuando el proceso de cambio de una configuración se produce por un paso de comunicación, cada nodo $x \in X_G$ envía una copia de cada palabra contenida en dicho nodo a todos los procesadores y conectados a x, teniendo en cuenta que ésta pueda atravesar el filtro de la arista entre x e y, y también

recibe todas las palabras enviadas por cualquier procesador z conectado con x que puedan atravesar el filtro de la arista entre x y z.

Formalmente, diremos que la configuración C' es obtenida en un paso de comunicación a partir de la configuración C, denotándolo como $C \vdash C'$, sí y sólo sí

$$C'(x) = (C(x) \setminus (\bigcup_{\{x,y\} \in E_G} \varphi^{\alpha(\{x,y\})}(C(x), \mathcal{N}(\{x,y\})))) \cup (\bigcup_{\{x,y\} \in E_G} \varphi^{\alpha(\{x,y\})}(C(y), \mathcal{N}(\{x,y\}))) \tag{3.12}$$

para todo $x \in X_G$.

Sea Γ un $ANSPFC$, el resultado de proceso de un Γ dada una palabra de entrada $z \in V^*$ por tanto será una secuencia de configuraciones $C_0^{(z)}, C_1^{(z)}, \cdots$, siendo $C_0^{(z)}$ la configuración inicial de Γ en z, $C_{2i}^{(z)} \Rightarrow C_{2i+1}^{(z)}$ y $C_{2i+1}^{(z)} \vdash C_{2i+2}^{(z)}$ para todo $i \geq 0$. Cada configuración resultante $C_i^{(z)}$ está unívocamente determinada por la configuración $C_{i-1}^{(z)}$. Por lo que se puede decir que cada computación en un $ANSPFC$ es determinista. La finalización de una computación se dice que es finita si se presenta una de estas dos condiciones:

- *Se produce una configuración en el nodo de salida x_O cuyo resultado es un conjunto de cadenas no vacío. Este tipo de resultado produce lo que denotaremos como computación por aceptación.*
- *Se repiten dos configuraciones idénticas como resultado de pasos consecutivos de splicing o comunicación.*

El lenguaje aceptado por Γ es:

$$L_a(\Gamma) = \{z \in V^* | \text{ la computación de } \Gamma \text{ sobre } z \text{ es por aceptación} \tag{3.13}$$

El lenguaje aceptado por Γ *con procesadores restrictivos es*

$$L_a^{(r)}(\Gamma) = \{z \in V^* | \text{ la computación de } \Gamma \text{ sobre } z \text{ es por aceptación} \quad (3.14)$$

Se dice que un $ANSPFC$ Γ *(formado con procesadores restrictivos) decide el lenguaje* $L \subseteq V^*$*, denotado por* $L(\Gamma) = L$ *si y sólo si* $L_a(\Gamma) = L$ *(* $L_a^{(r)}(\Gamma) = L$*) y la computación de* Γ *para cada* $z \in V^*$ *se detiene.*

Complejidad Temporal. *Definiremos dos medidas de complejidad computacional para la valoración de un* $ANSPFC$ *con o sin procesadores restrictivos como modelo computacional. Cosideraremos un* Γ *sobre el alfabeto de entrada* V *que se tiene en cada entrada. La complejidad en tiempo de la computación finita* $C_0^{(x)}, C_1^{(x)}, \cdots, C_m^{(x)}$ *de* Γ *sobre* $x \in V^*$ *se denota por* $Time_\Gamma(x)$ *y es igual a* m*. La complejidad temporal de* Γ *es la función parcial de* N *en* N*:*

$$Time_\Gamma = max\{Time_\Gamma(x) | x \in V^*, |x| = n\} \quad (3.15)$$

Se dice que Γ *decide* L *en tiempo* $O(f(n))$ *si* $Time_\Gamma(n) \in O(f(n))$*.*

Para una función $f : N \to N$ *se define:*

$$Time_{ANSPFC_p}(f(n)) = \{L | \exists\ ANSPFC\ \Gamma \text{ de tamaño } p\}$$

De tal manera que, Γ *decide* L *y* n_0 *tal que* $Time_\Gamma(n) \leq f(n) \forall n \geq n_0$*.*

Además, se denota $PTime_{ANSPFC_p} = \bigcup_{k \geq 0} Time_{ANSPFC_p}(n^k)$ *para todo* $p \geq 1$ *así como* $PTime_{ANSPFC} = \bigcup_{p \geq 1} PTime_{ANSPFC_p}$*.*

Complejidad en Tamaño. *Como tipo de medida adicional definiremos la complejidad en tamaño de la computación finita* $C_0^{(x)}, C_1^{(x)}, \cdots, C_m^{(x)}$ *de* Γ *sobre* $x \in L$ *denotada por* $Length_\Gamma(x)$ *y es igual a* $max_{w \in C_i^x(z), i \in \{1, \cdots, m\}, z \in X_G} |w|$*.*

Siendo Γ *La complejidad en tamaño, una función parcial de* N *en* N*:*

$$Legnth_{\Gamma}(n) = max\{Lenght_{\Gamma}(x)\ x \in V^*, |x| = n\} \tag{3.16}$$

Para una función $f : N \to N$ *se define:*

$$Length_{ANSPFC_p}(f(n)) = \{L|\exists\ ANSPFC\ \Gamma\ de\ tamaño\ p\}$$

De tal manera que Γ *decide* L *y* n_0 *tal que* $Length_{\Gamma}(n) \leq f(n) \forall n \geq n_0$*.*

Además, se denota $PLength_{ANSPFC_p} = \bigcup_{k\geq 0} Length_{ANSPFC_p}(n^k)$ *para todo* $p \geq 1$ *así como* $PLength_{ANSPFC} = \bigcup_{p\geq 1} PLength_{ANSPFC_p}$*.*

Las clases correspondientes de $ANSPFC$ *con procesadores restrictivos se denotan por* $PTime_{ANSPFC_p^{(r)}}$ *y* $PLength_{ANSPFC_p^{(r)}}$*.*

Conversión de NEP a un $ANSPFC$ equivalente

En este apartado formularemos el posible planteamiento que nos permitiría construir un $ANSPFC$ *equivalente a partir de un NEP y las restricciones que tendríamos que tener en cuenta para ello.*

Teorema 3.2.1. *Dada una red de procesadores evolutivos* $\Gamma^{(s)}$ *se puede construir una red de procesadores evolutivos con filtros en las conexiones* $\Gamma^{(s)}$ *equivalente.*

Demostración. Podremos obtener una red de procesadores evolutivos equivalente Γ^{-} con los filtros entre las conexiones de los procesadores.

Sea $\Gamma = (V, N_1, \cdots, N_n, G)$ una red de procesadores evolutivos de tamaño n con las condiciones de contexto definidas por los filtros:

$$\varphi^s(x;P,F) \equiv P \subseteq alph(x) \wedge F \cap alph(x) = \emptyset$$
$$\varphi^w(x;P,F) \equiv alph(x) \cap P \neq \emptyset \wedge F \cap alph(x) = \emptyset$$

De manera que cada procesador N_i dispone de sus filtros $PI_i, PO_i \in P$ y $FI_i, FO_i \in F$

En la conexión $e_{ij} \in E_G$ que conecta a los nodos N_i y N_j, pueden generarse los filtros situados en dicha conexión de la siguiente forma:

$$\mathcal{N}(e_{ij}) = (P_{e_{ij}}, F_{e_{ij}}) = (PO_i \cup PI_j, FO_i \cup FI_j) \tag{3.17}$$

de esta manera,

$$\varphi^s(x;\mathcal{N}(e_{ij})) \equiv \varphi^s(x;P_{e_{ij}},F_{e_{ij}}) \equiv \varphi^s(x;PO_i \cup PI_j, FO_i \cup FI_j)$$

Este supuesto no tendrá validez en el caso de que los filtros se comporten en su forma débil $\Gamma^{(w)}$ ya que:

$$\varphi^w(x;\mathcal{N}(e_{ij})) \equiv \varphi^w(x;P_{e_{ij}},F_{e_{ij}}) \equiv \varphi^w(x;PO_i \cup PI_j, FO_i \cup FI_j)$$

no siendo equivalente a:

$$\varphi^w(x;PO_i,FO_i) \wedge \varphi^w(x;PI_j,FI_j)$$

De esta manera se puede construir una red $\Gamma^{(s)}$ equivalente pero con los filtros situados en las conexiones entre los procesadores. □

3.3. Resolución de problemas con *ANSPFC*

A continuación se describe cómo se pueden emplear los ANSPFC para resolver problemas. Una posible correspondencia entre problemas de decisión y lenguajes se puede efectuar a través de una función que transforma una instancia de un problema de decisión dado en una palabra [29]. Un problema de decisión P se resuelve en tiempo $O(f(n))$ por un ANSPFC si existe una familia $\mathcal{G}$ de ANSPFC tal que se cumplen las siguientes condiciones:

1. *La función de codificación de cualquier instancia p de P con tamaño n se puede computar empleando una máquina de Turing determinista en tiempo $O(f(n))$.*

2. *Para cada instancia p de tamaño n del problema, se puede construir, en tiempo $O(f(n))$, un $ANSPFC$ $\Gamma(p) \in \mathcal{G}$ que decide, en tiempo $O(f(n))$, la palabra que codifica la correspondiente instancia. Esto quiere decir que la palabra es aceptada si y sólo si la solución a la instancia dada del problema es "SI".*

Si un $ANSPFC$ $\Gamma \in \mathcal{G}$ construido como se ha descrito decide el lenguaje de palabras que codifican todas las instancias del mismo tamaño n, entonces la construcción de Γ se denomina solución uniforme. Una solución es uniforme si para un problema de tamaño n se puede construir un único $ANSPFC$ que resuelve todas las instancias de tamaño n tomando como entrada la codificación de la instancia.

Seguidamente se presenta una solución uniforme lineal en tiempo a un problema NP: SAT (satisfactibilidad). Este es el problema original NP-completo. Una instancia del SAT consiste en una fórmula E con n variables y m cláusulas. La fórmula E es una conjunción de m cláusulas y cada una de ellas es la disyunción de varias variables o la negación de un conjunto de n variables. Se asume que cada variable o su negación aparece en al menos una cláusula. El problema consiste en decidir si existe o no una asignación de las n variables booleanas tal que las m cláusulas son todas satisfechas.

Teorema 3.3.1. *SAT se puede resolver uniformemente en tiempo lineal por un ANSFPC. El tamaño, número de símbolos, reglas y palabras auxiliares del ANSPFC que resuelve el SAT están linealmente acotados por el tamaño de la instancia del SAT.*

Demostración. Sea V el conjunto de variables, $V = \{x_1, x_2, \cdots, x_n\}$ y $\phi = (C_1) \wedge (C_2) \wedge \cdots (C_m)$ es la fórmula booleana, donde la negación de la variable x_i se denota por $\bar{x}_i$. Cada fórmula se puede ver como una palabra sobre el alfabeto $U = V \cup \bar{V} \cup \{\wedge, \vee, (,)\}$, donde $\bar{V} = \{\bar{x} | x \in V\}$.
Se define el alfabeto:

$$W = \{\lhd x_i = 1 \rhd, \lhd x_i = 0 \rhd | 1 \leq i \leq n\} \cup U \cup \{\#, \uparrow, >, >_1\}$$

Sea el $ANSPFC$:

$$\Gamma = (U, W, <, >, K_{2n+2}, \mathcal{N}, \alpha, In, Out)$$

donde K_{2n+2} es el grafo completo con $2n + 2$ nodos, In, Out, $(x_i \leftarrow 1)$, $(x_i \leftarrow 0) 1 \leq i \leq n$ siendo:

- In:
  - $S_{In} = \{[<, (; \uparrow \lhd x_1 = b \rhd, \#] | b \in \{0, 1\}\} \cup \{[\uparrow, \lhd x_i = b \rhd; \uparrow \lhd x_{i+1} = b' \rhd, \#]\ i \in \{1, \cdots, n-2\}, b, b' \in \{0, 1\}\} \cup \{[\uparrow, \lhd x_{n-1} = b \rhd; < \lhd x_n = b' \rhd, \#] | b, b' \in \{0, 1\}\}$
  - $A_{In} = \{\uparrow \lhd x_i = b \rhd \# | b \in \{0, 1\}, i \in \{1, \cdots, n-1\}\} \cup \{< \lhd x_n = b \rhd \# | b \in \{0, 1\}\}$
- Out:
  - $S_{Out} = A_{Out} = \emptyset$
- $(x_i \leftarrow 1)$:

- $S_{(x_i \leftarrow 1)} = \{[\epsilon, \vee x_i) >; \#,) >_1], [\epsilon, \wedge(x_i >; \#, >], [\epsilon, \vee \bar{x}_i >; \#,) >], [\epsilon, \wedge(\bar{x}_i) >; \#, \uparrow]\}$
- $A_{(x_i \leftarrow 1)} = \{\#) >, \#) >_1, \# >, \# \uparrow\}$

- $(x_i \leftarrow 0)$:
  - $S_{(x_i \leftarrow 0)} = \{[\epsilon, \vee \bar{x}_i) >; \#,) >_1], [\epsilon, \wedge(\bar{x}_i >; \#, >], [\epsilon, \vee x_i >; \#,) >], [\epsilon, \wedge(x_i) >; \#, \uparrow]\}$
  - $A_{(x_i \leftarrow 0)} = \{\#) >, \#) >_1, \# >, \# \uparrow\}$

- $\mathcal{N}(\{In, (x_i \leftarrow b)\}) = (\{\lhd\, x_i = b \,\rhd\}, \{\uparrow, \#\})$,
  $\alpha(\{In, (x_i \leftarrow b)\}) = (s), 1 \leq i \leq n, b \in \{0, 1\}$

- $\mathcal{N}(\{(x_i \leftarrow a), (x_j \leftarrow b)\}) = (W, \{\uparrow, \#\})$,
  $\alpha(\{(x_i \leftarrow a), (x_j \leftarrow b)\}) = (w), 1 \leq i \leq n, a, b \in \{0, 1\}$

- $\mathcal{N}(\{(x_i \leftarrow a), Out\}) = (\{\lhd\, x_j \leftarrow b \,\rhd\, | 1 \leq j \leq n, b \in \{0, 1\}, U \cup \{\uparrow, \#\})$,
  $\alpha(\{(x_i \leftarrow a), Out\}) = (w), 1 \leq i \leq n, a \in \{0, 1\}$

Claramente, dado un n el $ANSPFC\ \Gamma$ se puede construir en tiempo $O(n)$. Ahora, dada una instancia del SAT sobre n variables, esto es una fórmula $\phi = (C_1) \wedge (C_2) \wedge \cdots \wedge (C_m)$ para algún $m \geq 1$, se definirá la palabra que codifica esta instancia como ϕ.

A continuación se discute como Γ trabaja en la palabra de entrada ϕ. Se asume que no existen dos cláusulas idénticas en ϕ. El algoritmo implementado por esta red es el siguiente: todos los posibles valores se asignan a las variables en el nodo In, y se calcula la fórmula de derecha a izquierda. En la configuración inicial la palabra $< \phi >$ está en el nodo In. En los siguientes $2n - 1$ pasos de computación, de los cuales n son de *splicing*, no se comunica ninguna palabra ya que ninguna puede abandonar el nodo In. Más concretamente, después de k pasos de *splicing* todas las palabras:

$$\uparrow \lhd\, x_k = b_k \,\rhd \cdots \lhd\, x_1 = b_1 \,\rhd\, \phi >$$

con $b_j \in \{0,1\}, 1 \leq j \leq k$ están en In. Debido a la aparición del símbolo $\uparrow$, ninguna de estas palabras puede abandonar In hasta que $k = n$. Por tanto, después de $2n-1$ pasos cada una de estas palabras contendrán $\lhd x_n = 1 \rhd$ ó $\lhd x_n = 0 \rhd$ y por consiguiente pueden atravesar los filtros de todas las aristas incidentes sobre In y comunicadas sobre otros nodos. Por disponer los filtros en las aristas de Γ una copia de cada palabra $<\lhd x_n = b_n \rhd \cdots \lhd x_1 = b_1 \rhd \phi >$ se recibe en todos los nodos $(x_1 \leftarrow b_1), \cdots, (x_n \leftarrow b_n)$.

A partir de ahora, las palabras obtenidas a partir de operaciones de *splicing* sobre la palabra original saldrán y entrarán en In (donde las operaciones de *splicing* no tendrán efecto), y $(x_1 \leftarrow b_1), \cdots, (x_n \leftarrow b_n)$ por lo menos en $2m$ pasos de *splicing*. Al mismo tiempo, las copias de estas palabras estarán continuamente moviéndose entre cualquier par de nodos del conjunto $\{(x_1 \leftarrow b_1), \cdots, (x_n \leftarrow b_n)\}$ durante $2m$ fases de *splicing*.

Se asume que ϕ es satisfactible, esto es, existe una asignación de variables que satisface cada cláusula. Sea $x_i = b_i, b_i \in \{0,1\}, 1 \leq i \leq n$ tal asignación. Entonces, para cada paso de *splicing*, el símbolo de más a la derecha x_i o $\bar{x}_i$, para algún $1 \leq i \leq n$, de todas las palabras que entran en el nodo $(x_i \leftarrow b_i)$ se elimina. Nótese que si tal palabra entra en un nodo $(x_j \leftarrow b_j)$, $i \neq j$, no se modifica en absoluto. Ya que cada cláusula es una disyunción, para cada $1 \leq k \leq m$, existe $1 \leq i_k \leq n$ tal que $x_{i_k} = b_{i_k}$ satisface la cláusula C_k. Además, se asume que x_{i_k} es la variable de más a la derecha que aparece en C_k y satisface C_k. Cuando x_{i_k} es el símbolo de más a la derecha de una fórmula, se aplica una regla del nodo $(x_{i_k} \leftarrow b_{i_k})$, reemplazando:

- o bien, $\vee x_{i_k}$) por) $>_1$ siempre y cuando $b_{i_k} = 1$.
- o bien, $\vee \bar{x}_{i_k}$) por) $>_1$ siempre y cuando $b_{i_k} = 0$.
- o bien, $\wedge x_{i_k}$) por) $>_1$ siempre y cuando $b_{i_k} = 1$.
- o bién, $\wedge \bar{x}_{i_k}$) por) $>_1$ siempre y cuando $b_{i_k} = 0$.

Se interpreta $>_1$ como un marcador, significando que la cláusula actual se satisface. Como se puede ver, el proceso continua con $>$ para cualquier cláusula. Por tanto, después de m pasos de *splicing* se genera la palabra

$<\!\!\lhd x_1 = b_1 \rhd\!\lhd x_2 = b_2 \rhd \cdots \lhd x_n = b_n \rhd\!\!>$ que entra en Out, sólo palabras de esa forma pueden entrar en Out.

Por otro lado, es fácil de ver que si Γ acepta una palabra ϕ, entonces cualquier cláusula de ϕ se satisface, y por tanto, ϕ es satisfactible. De estas explicaciones se deduce que Γ acepta cualquier palabra ϕ en tiempo lineal si y sólo si ϕ es satisfactible.

Para terminar la demostración, se ve que Γ se detiene en cualquier palabra ϕ donde ϕ no es satisfactible. Para cualquier posible asignación, existe una cláusula que no se satisface por tal asignación. Sea C_{s_i} la cláusula de más a la derecha de ϕ que no se satisface con la asignación i, $1 \leq i \leq 2^n$. Cuando se alcanza el conector $\wedge$ antes de C_{s_i}, el símbolo $\uparrow$ se introduce como marcador por la derecha, después no se puede aplicar ninguna regla de *splicing* y tampoco se puede enviar nunca más. En el peor caso, $C_{s_i} = C_1$ y Γ se detiene en ϕ en $O(n+m)$ pasos. □

Notesé que los otros recursos en Γ *están linealmente acotados:*

- *El número de nodos es* $2n + 2$.
- *El número total de símbolos de* W *es* $4n + 8$.
- *El número total de reglas de splicing es* $12n - 2$.
- *El número total de axiomas es* $10n$.

En este capítulo se ha propuesto un modelo de Redes de Procesadores Evolutivos con Filtros en las Conexiones, y concretamente una variante ANSPFC (Accepting Networks of Evolutionary Processors with Filtered Connections) que resuelve en tiempo uniforme lineal el problema SAT. Esta solución se ha de entender en el siguiente contexto: no se resuelve el problema SAT con recursos lineales de tiempo y espacio, ya que tanto las palabras del alfabeto y palabras auxiliares están presentes con un número ilimitado de ocurrencias. Sin embargo, esto no es un gran inconveniente ya que a través de la operación de PCR (Polymerase Chain Reaction), es posible generar un número

exponencial de moléculas de ADN idénticas empleando un número lineal de reacciones, veáse [14].

Cabe reseñar que el modelo ANSPFC descrito a lo largo de este capítulo posee una estructura fija para cualquier instancia del problema siempre y cuando posea el mismo número de variables. Por tanto, la solución es uniforme en el sentido de que la red, excepto los nodos de entrada y salida, se puede ver como un algoritmo: teniendo en cuenta el número de variables, se definen los filtros, las palabras y reglas de splicing y se asignan todos los posibles valores a las variables, evaluándose la fórmula.

En [14] se demuestra que un ANSP es computacionalmente completo. De lo que cabe preguntar, dejando para unas posibles líneas de investigación que continúen con el trabajo realizado en esta tesis: Si un ASNPFC será computacionalmente completo. Si este no es el caso, ¿qué tipo de problemas se pueden resolver eficientemente empleando el modelo descrito?. Además, el tipo de problemas NP es exactamente igual al tipo de todos los lenguajes decidibles por un ANSP en tiempo polinomial. Después de lo anteriormente expuesto convendría profundizar en una futura línea de investigación sobre la siguiente cuestión: ¿Se pueden caracterizar los ANSPFC del mismo modo que los ANSP?.

Parte IV

Simulaciones

Capítulo 4

Modelización del simulador

El Lenguaje elegido para el desarrollo de la herramienta de simulación es Java y en concreto se va a utilizar los threads de Java para ejecutar múltiples procesos mediante procesos ligeros. No es el objetivo principal de esta tesis mostrar en ella todos los procesos y procedimientos de Ingeniería del Software utilizados a la hora de desarrollar la herramienta del simulador, por lo que se realizará una descripción de los aspectos más destacados del mismo con la finalidad de poder tener una visión genérica de la arquitectura del simulador y de sus principales características utilizadas para la implementación de los procesadores de un NEP. En la figura 4.1 se muestra el diagrama de clases donde se puede apreciar la existencia de las siguientes clases:

- *Processor, que incluye todos los elementos necesarios para el comportamiento esperado del procesador.*
- *Rules encargada de la implementación y comportamiento de las reglas aplicadas al procesador*
- *Filter, encargada de los filtros del procesador.*
- *InputFilter y OutputFilter, describen a su vez los filtros de entrada y salida del procesador.*
- *Obj, los objetos que suministran la entrada del procesador.*

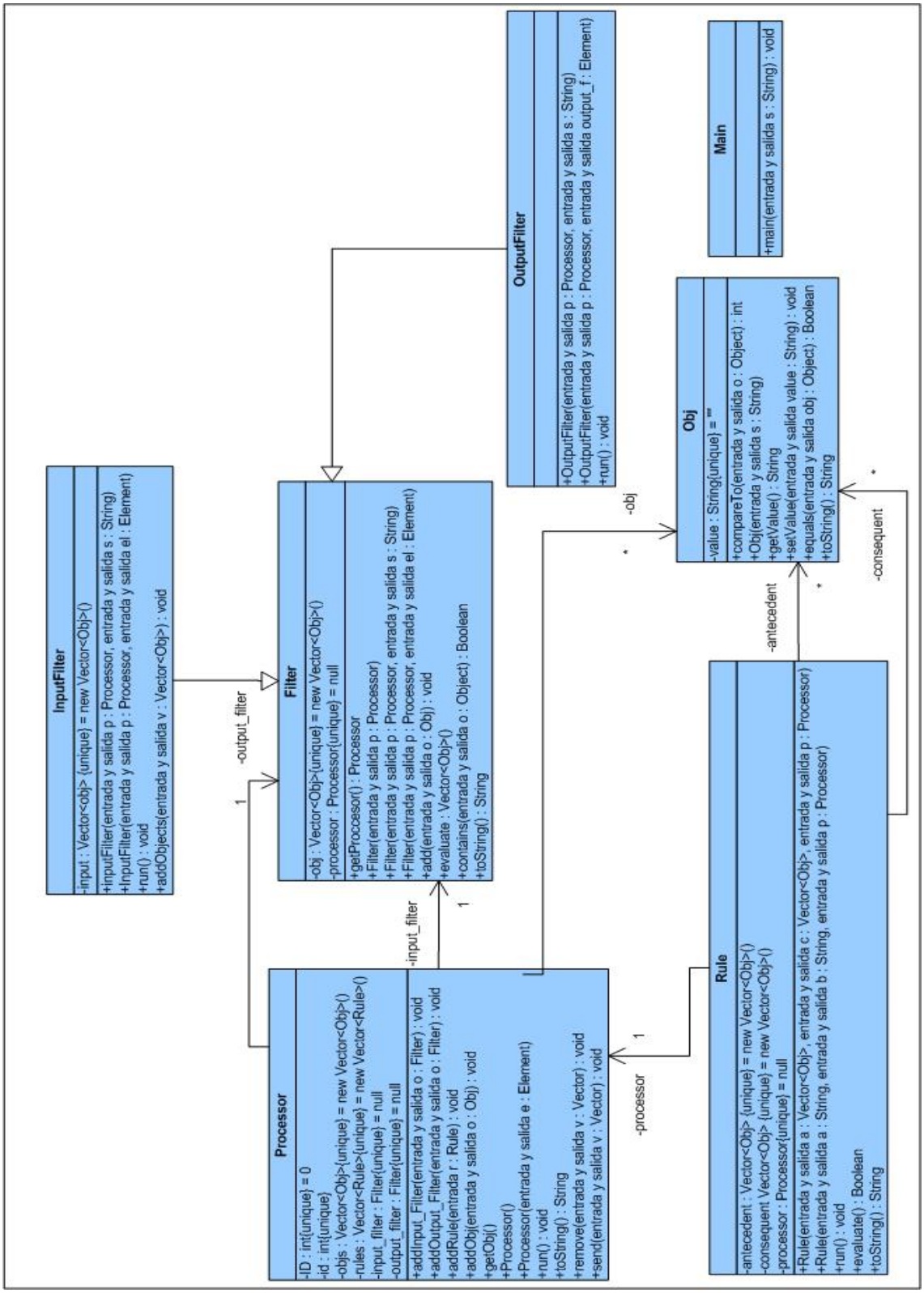


Figura 4.1: Diagrama UML de clases.

A continuación se realiza una descripción de como es simulado el comportamiento de los procesadores en el simulador: Los procesadores de un NEP se deberían comportar de una manera no determinista. Los cambios en una configuración se deben a pasos de comunicación y evolución, pero estos dos pasos tienen lugar de manera independiente, esto es, evolución o comunicación se escogen dependiendo del modelo de threads implementado en [54, 53, 56]. Las reglas y filtros (de entrada y salida) se implementan como threads que extienden el interface Runnable aportado por Java. Por tanto, un procesador es el padre de un conjunto de threads, que emplean los objetos del procesador en una región de exclusión mútua. Las reglas, filtros y objetos son parte de un procesador. Los filtros pueden ser de entrada o salida, dependiendo de su comportamiento, y controlan como los objetos se envían y reciben por diferentes procesadores. Las reglas de sustitución tienen un antecedente y un consecuente implementado como un conjunto de objetos. Cuando un procesador se ejecuta a través del método start, arranca en modo cascada el thread de los filtros y reglas.

Esta es la composición básica de un procesador evolutivo; sin embargo, existen arquitecturas NEP que tienen filtros prohibitivos en la entrada y salida. Las diferencias en la implementación para la resolución de problemas se definirá como tipos del modelo genérico para tales clases de problemas.

De acuerdo con la figura 4.1, cada procesador tiene un número de reglas, objetos y un filtro de entrada y salida. Cuando el procesador arranca todas las reglas y filtros comienzan como threads independientes, véase figura 4.2 y el listado 4.3. Los objetos en el procesador se almacenan empleando la clase Vector que es thread-safe, por tanto la sincronización está garantizada.

Funcionamiento de un procesador.

Listado 4.1: Comportamiento de un Procesador

```
1 public void run() {
2   for (int i=0; i<this.rules.size(); i++)
3    new Thread(this.rules.get(i)).start();
4   new Thread((OutputFilter) this.output_filter).start();
5   new Thread((InputFilter) this.input_filter).start();
6   return;
7 }
```

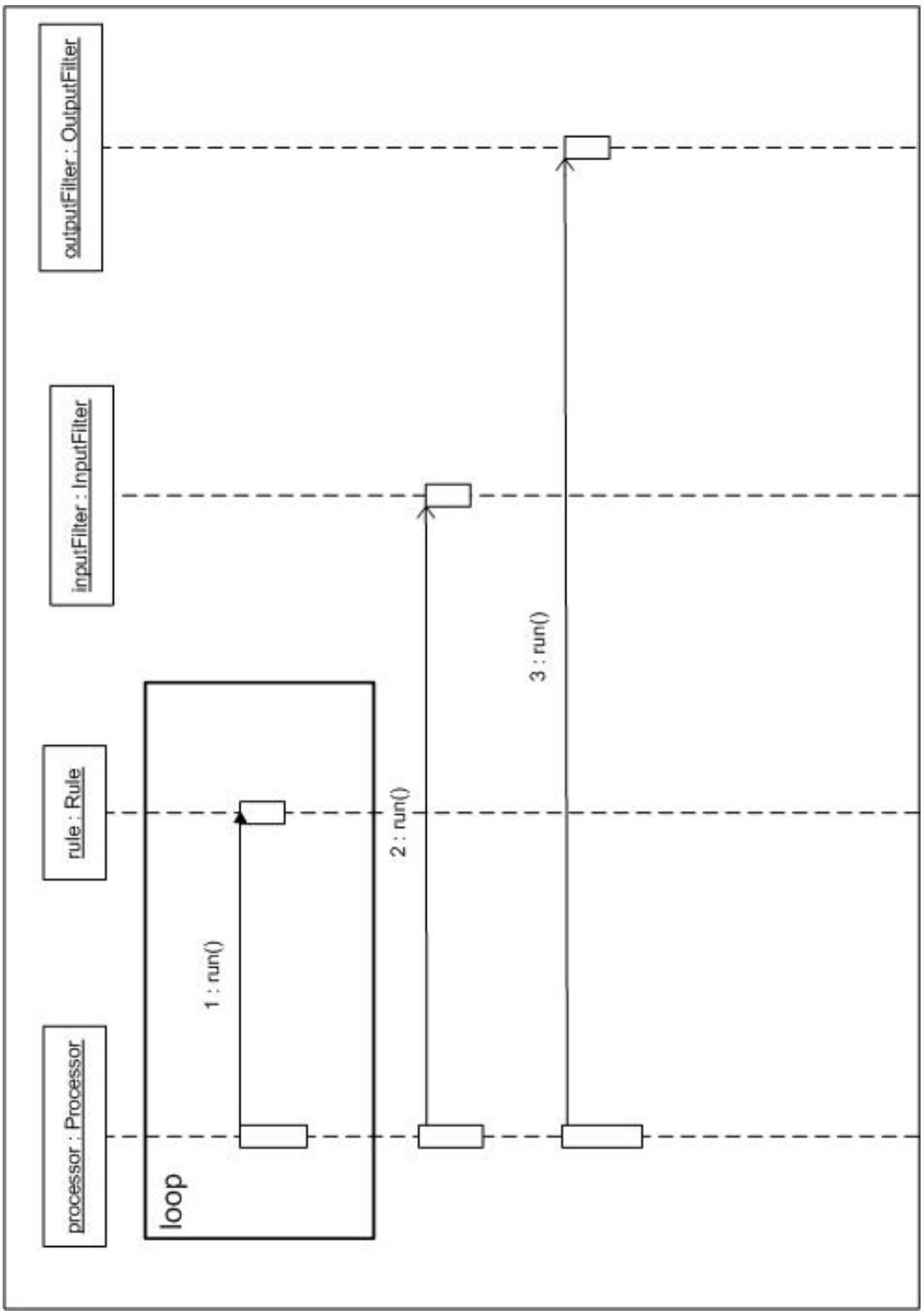


Figura 4.2: Diagrama de secuencia de un procesador.

Los procesadores pueden enviar y recibir objetos siempre y cuando las restricciones de los filtros se satisfagan. Esta comunicación se realiza de la manera siguiente:

- *Envío de objetos. Los filtros de salida comprueban restricciones y todos los objetos que cumplen las restricciones se eliminan del pool de objetos.*

Listado 4.2: Envío de objetos

```
1 public void run() {
2     Vector v = null;
3      while (true) {
4       v = super.evaluate();
5       super.getProccesor().remove(v);
6       super.send(v);
7      }
8   }
```

- *Recibir objetos. Cuando el método de envío se invoca desde otro procesador, algunos objetos ubicados en el pool de objetos del filtro de entrada se comprueban, si cumplen las restricciones entonces se añaden.*

Listado 4.3: Recepción de objetos

```
1 synchronized public void send(Vector v) {
2     ((InputFilter)this.input_filter).addObjects(v);
3 }
```

Todas las reglas y filtros son tratados como threads, por tanto el comportamiento no determinista está garantizado [52].

Los threads asociados a las reglas son muy simples, tan sólo comprueban si los objetos del antecedente están en el pool del procesador, si esto es así los objetos del consecuente se incorporan en dicho pool, vease figura 4.3. No hay ningún orden en la aplicación de las reglas, éstas están en threads separados, por tanto todas las comprobaciones sobre el pool de objetos se pueden realizar al mismo tiempo y se pueden aplicar al mismo tiempo Se ha introducido un retraso aleatorio para obtener una simulación más realista. No se han considerado reglas de inserción y borrado en nuestro sistemas, pero se pueden añadir fácilmente definiendo un objeto null en el fichero de configuración

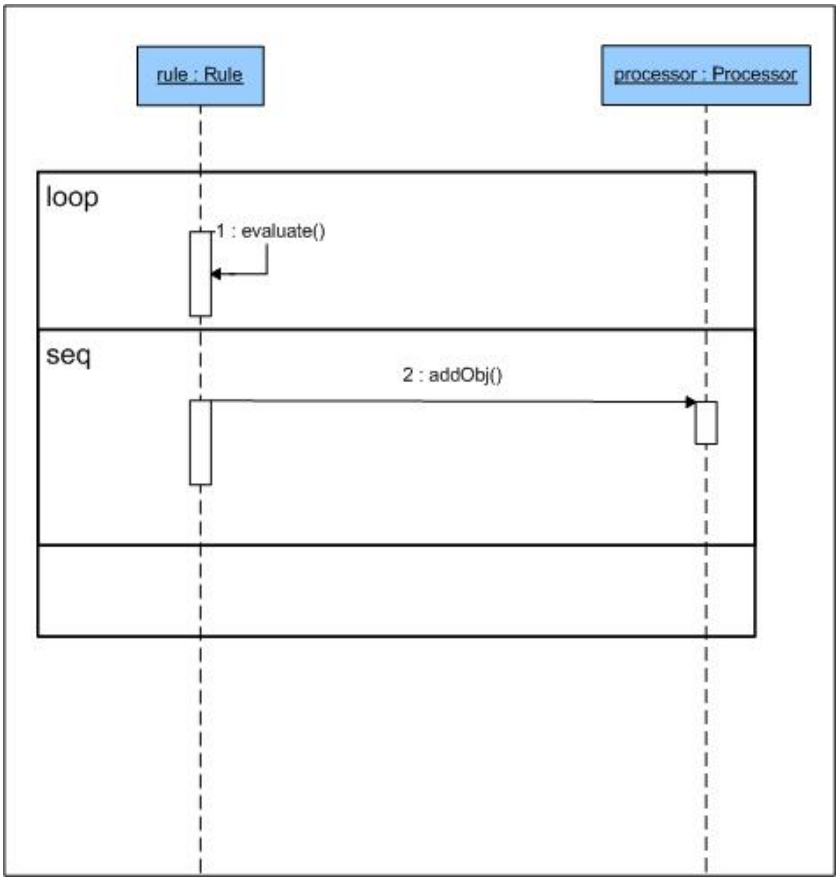


Figura 4.3: Diagrama de secuencia de una regla.

config.xml(donde se configura el simulador) y el sistema funcionará teniendo en cuenta tal esquema.

El listado 4.4 muestra un esquema del comportamiento de las reglas de acuerdo con la figura 4.3. Se puede notar que si el antecedente es satisfactoriamente evaluado entonces todos los objetos del consecuente se añadirán al pool del procesador.

Listado 4.4: Comportamiento de las reglas

```
1 while(true) {
2  if (this.evaluate()) {
3   Enumeration<Obj> e = this.consequent.elements();
4   while (e.hasMoreElements()) {
5    Obj o = e.nextElement();
6    if (!processor.getObj().contains(o)) this.processor.addObj(o);
7   }
8  }
9 }
```

Los filtros trabajan en paralelo con las reglas. Cuando las restricciones de los filtros se cumplen entonces se eliminarán o añadirán algunos objetos al pool del procesador. La principal diferencia entre los filtros de entrada y salida es, véase las figuras 4.4 y 4.5:

- *Los filtros de entrada tan sólo añaden objetos al procesador si estos cumplen las restricciones.*
- *Los filtros de salida evaluan el pool de objetos para descubrir qué objetos se deben enviar.*

Ambos emplean la funcionalidad de una clase Filter, que proporciona la evaluación de objetos. Se pueden implementar diferentes tipos de filtros en un procesador evolutivo. Un filtro es un sistema que permite a un símbolo ir de un procesador a otro. Normalmente, el sistema de detección consiste en comparar un símbolo con otro.

El listado 4.5 muestra el método de evaluación que corresponde al filtro de salida que comprueba si los objetos presentes en el pool se pueden enviar o no a los procesadores conectados, si es así, se eliminarán del procesador.

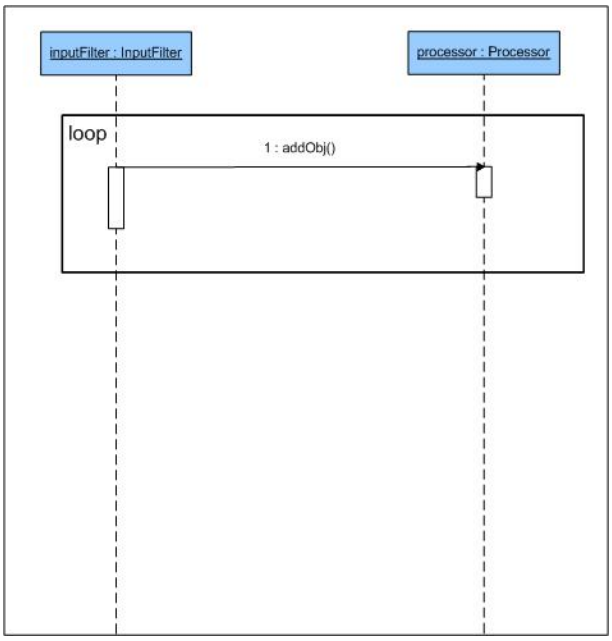


Figura 4.4: Diagrama de secuencia del filtro de entrada.

Listado 4.5: Evaluación del filtro de salida

```
1  public Vector<Obj> evaluate() {
2   Vector<Obj> v = this.processor.getObj();
3   Enumeration<Obj> e = v.elements();
4   v = new Vector<Obj>();
5   while(e.hasMoreElements()) {
6    Obj o = e.nextElement();
7    if (this.obj.contains(o)) if (!v.contains(o)) v.add(o);
8   }
9   return v;
10 }
```

Se está realizando un gran esfuerzo en la definición teórica de diferentes familiar de NEPs y el estudio de sus propiedades formales, así como su completitud computacional y su capacidad de resolver problemas NP en tiempo polinomial. Sin embargo, a parte del enfoque proporcionado por esta tesis y el trabajo posterior de [25], no se ha realizado ningún tipo de simulación real o implementación de los modelos basados en NEP. El simulador propuesto es un primer modelo de implementación de la familia de dispositivos de computación simbólica denominados NEPs. Podría ser mejorando adaptándolo para

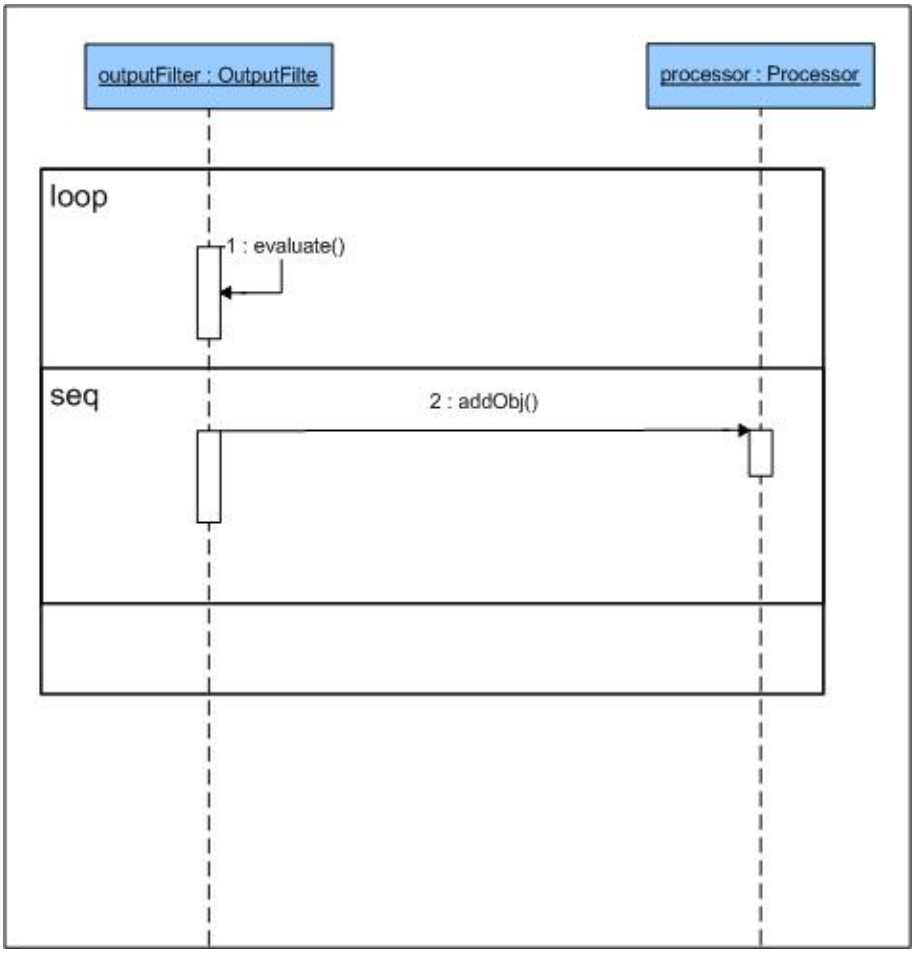


Figura 4.5: Diagrama de secuencia del filtro de salida.

su ejecución en Cluster de Ordenadores, dando soporte Web y desarrollando un Framework de trabajo para desarrollar sobre el mismo. jNEP [25] es un programa Java que es capaz de simular cualquier NEP clásico descrito en la literatura. Ha sido diseñado con dos principios básicos:

- *Sigue rigurosamente la definición formal de los NEP, en cuanto a paralelismo y distribución.*
- *Es una herramienta general, permitiendo el uso de diferentes variantes de NEP y se puede adaptar a futuras variantes (al igual que las descritas en este trabajo).*

El código de esta implementación está disponible en la siguiente dirección: `http://jnep.e-delrosal.net`.

4.1. Simulación y Resultados de los *NEP*s

El listado 4.6 muestra como se define la arquitectura y configuración inicial de una red de procesadores evolutivos, concretamente se muestra la configuración inicial para el caso del problema de los 3 *colores. Hay que tener en cuenta que el ejemplo mostrado se simula sobre una red de procesadores evolutivos masivamente paralela.*

Listado 4.6: Configuración Inicial – Fichero `conf.xml`

```
<?xml version="1.0"?>
<NEP>
 <processor>
  <name>0</name>
  <object>acde</object>
  <rule>
   <antecedent>
    <object>a</object>
   </antecedent>
   <consequent>
    <object>rA</object>
    <object>gA</object>
    <object>bA</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>c</object>
   </antecedent>
   <consequent>
    <object>rC</object>
    <object>gC</object>
    <object>bC</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>d</object>
   </antecedent>
   <consequent>
    <object>rD</object>
    <object>gD</object>
    <object>bD</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>e</object>
   </antecedent>
   <consequent>
    <object>rE</object>
    <object>gE</object>
    <object>bE</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>Vacio</object>
  </inputfilter>
  <outputfilter>
   <object>A</object>
   <object>C</object>
   <object>D</object>
   <object>E</object>
  </outputfilter>
 </processor>
 <processor>
  <name>1</name>
  <rule>
   <antecedent>
    <object>rA</object>
   </antecedent>
   <consequent>
    <object>ra</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>gA</object>
   </antecedent>
   <consequent>
    <object>ga</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>rC</object>
   </antecedent>
   <consequent>
    <object>rc</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>gC</object>
   </antecedent>
   <consequent>
    <object>gc</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>A</object>
   <object>C</object>
  </inputfilter>
  <outputfilter>
   <object>g</object>
   <object>r</object>
  </outputfilter>
 </processor>
 <processor>
  <name>2</name>
  <rule>
   <antecedent>
    <object>XGA</object>
   </antecedent>
   <consequent>
```

```
    <object>ga</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>XBA</object>
   </antecedent>
   <consequent>
    <object>ba</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>YGC</object>
   </antecedent>
   <consequent>
    <object>gc</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>YBC</object>
   </antecedent>
   <consequent>
    <object>bc</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>A</object>
  </inputfilter>
  <outputfilter>
   <object>b</object>
   <object>g</object>
  </outputfilter>
 </processor>
 <processor>
  <name>2</name>
  <rule>
   <antecedent>
    <object>XRA</object>
   </antecedent>
   <consequent>
    <object>ra</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>XBA</object>
   </antecedent>
   <consequent>
    <object>ba</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>YRC</object>
   </antecedent>
   <consequent>
    <object>rc</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>YBC</object>
   </antecedent>
   <consequent>
    <object>bc</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>A</object>
  </inputfilter>
  <outputfilter>
   <object>r</object>
   <object>b</object>
  </outputfilter>
 </processor>
 <processor>
  <name>4</name>
  <inputfilter>
   <object>a</object>
   <object>c</object>
  </inputfilter>
  <outputfilter>
  </outputfilter>
 </processor>
 <processor>
  <name>5</name>
  <rule>
   <antecedent>
    <object>TBE</object>
   </antecedent>
   <consequent>
    <object>be</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>TGE</object>
   </antecedent>
   <consequent>
    <object>ge</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>ra</object>
  </inputfilter>
  <outputfilter>
```

```
    <object>e</object>
   </outputfilter>
  </processor>
  <processor>
   <name>6</name>
   <rule>
    <antecedent>
     <object>TRE</object>
    </antecedent>
    <consequent>
     <object>re</object>
    </consequent>
   </rule>
   <rule>
    <antecedent>
     <object>TBE</object>
    </antecedent>
    <consequent>
     <object>be</object>
    </consequent>
   </rule>
   <inputfilter>
    <object>ga</object>
   </inputfilter>
   <outputfilter>
    <object>e</object>
   </outputfilter>
  </processor>
  <processor>
   <name>7</name>
   <rule>
    <antecedent>
     <object>TRE</object>
    </antecedent>
    <consequent>
     <object>re</object>
    </consequent>
   </rule>
   <rule>
    <antecedent>
     <object>TGE</object>
    </antecedent>
    <consequent>
     <object>ge</object>
    </consequent>
   </rule>
   <inputfilter>
    <object>ba</object>
   </inputfilter>
   <outputfilter>
    <object>e</object>
   </outputfilter>
  </processor>
  <processor>
   <name>8</name>
   <rule>
    <antecedent>
     <object>ZRD</object>
    </antecedent>
    <consequent>
     <object>rd</object>
    </consequent>
   </rule>
   <rule>
    <antecedent>
     <object>ZGD</object>
    </antecedent>
    <consequent>
     <object>gd</object>
    </consequent>
   </rule>
   <rule>
    <antecedent>
     <object>ZBD</object>
    </antecedent>
    <consequent>
     <object>bd</object>
    </consequent>
   </rule>
   <inputfilter>
   </inputfilter>
   <outputfilter>
    <object>d</object>
   </outputfilter>
  </processor>
  <processor>
   <name>9</name>
   <rule>
    <antecedent>
     <object>ba</object>
    </antecedent>
    <consequent>
     <object>bA</object>
    </consequent>
   </rule>
   <rule>
    <antecedent>
     <object>ga</object>
    </antecedent>
    <consequent>
     <object>gA</object>
    </consequent>
   </rule>
   <rule>
    <antecedent>
     <object>d</object>
    </antecedent>
    <consequent>
```

```
    <object>D</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>rd</object>
  </inputfilter>
  <outputfilter>
   <object>D</object>
   <object>A</object>
  </outputfilter>
 </processor>
 <processor>
  <name>10</name>

  <rule>
   <antecedent>
    <object>ba</object>
   </antecedent>
   <consequent>
    <object>bA</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>ra</object>
   </antecedent>
   <consequent>
    <object>rA</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>d</object>
   </antecedent>
   <consequent>
    <object>D</object>
   </consequent>
  </rule>

  <inputfilter>
   <object>gd</object>
  </inputfilter>
  <outputfilter>
   <object>D</object>
   <object>A</object>
  </outputfilter>
 </processor>
 <processor>
  <name>11</name>
  <rule>
   <antecedent>
    <object>ga</object>
   </antecedent>
   <consequent>
    <object>gA</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>ra</object>
   </antecedent>
   <consequent>
    <object>rA</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>d</object>
   </antecedent>
   <consequent>
    <object>D</object>
   </consequent>
  </rule>
  <inputfilter>
   <object>bd</object>
  </inputfilter>
  <outputfilter>
   <object>D</object>
   <object>A</object>
  </outputfilter>
 </processor>

 <processor>
  <name>12</name>
  <rule>
   <antecedent>
    <object>A</object>
   </antecedent>
   <consequent>
    <object>a</object>
   </consequent>
  </rule>
  <inputfilter>
  </inputfilter>
  <outputfilter>
   <object>a</object>
  </outputfilter>
 </processor>
 <processor>
  <name>13</name>
  <rule>
   <antecedent>
    <object>bD</object>
   </antecedent>
   <consequent>
    <object>bd</object>
   </consequent>
  </rule>
```

```
  <rule>
   <antecedent>
    <object>gD</object>
   </antecedent>
   <consequent>
    <object>gd</object>
   </consequent>
  </rule>

  <inputfilter>
   <object>rc</object>
  </inputfilter>
  <outputfilter>
   <object>c</object>
   <object>d</object>
  </outputfilter>
 </processor>
 <processor>
  <name>14</name>
  <rule>
   <antecedent>
    <object>bD</object>
   </antecedent>
   <consequent>
    <object>bd</object>
   </consequent>
  </rule>
  <rule>
   <antecedent>
    <object>rD</object>
   </antecedent>
   <consequent>
    <object>rd</object>
   </consequent>
  </rule>

  <inputfilter>
   <object>gc</object>
  </inputfilter>
  <outputfilter>
   <object>c</object>
   <object>d</object>
  </outputfilter>
 </processor>
 <processor>
  <name>15</name>
  <rule>
   <antecedent>
    <object>gD</object>
   </antecedent>
   <consequent>
    <object>gd</object>
   </consequent>
  </rule>
```

```
  <rule>
   <antecedent>
    <object>rD</object>
   </antecedent>
   <consequent>
    <object>rd</object>
   </consequent>
  </rule>

  <inputfilter>
   <object>bc</object>
  </inputfilter>
  <outputfilter>
   <object>c</object>
   <object>d</object>
  </outputfilter>
 </processor>


 <processor>
  <name>16</name>
  <inputfilter>
  </inputfilter>
  <outputfilter>
   <object>OO</object>
  </outputfilter>
 </processor>
 <conn>
  <from>0</from>
  <to>1</to>
 </conn>
 <conn>
  <from>0</from>
  <to>2</to>
 </conn>
 <conn>
  <from>0</from>
  <to>3</to>
 </conn>
 <conn>
  <from>3</from>
  <to>4</to>
 </conn>
 <conn>
  <from>2</from>
  <to>4</to>
 </conn>
 <conn>
  <from>1</from>
  <to>4</to>
 </conn>
 <conn>
  <from>4</from>
  <to>5</to>
```

```
</conn>
<conn>
 <from>4</from>
 <to>6</to>
</conn>
 <conn>
 <from>4</from>
 <to>7</to>
</conn>
<conn>
 <from>5</from>
 <to>8</to>
</conn>
<conn>
 <from>6</from>
 <to>8</to>
</conn>
<conn>
 <from>7</from>
 <to>8</to>
</conn>
<conn>
 <from>8</from>
 <to>9</to>
</conn>
<conn>
 <from>8</from>
 <to>10</to>
</conn>
<conn>
 <from>8</from>
 <to>11</to>
</conn>
<conn>
 <from>9</from>
 <to>12</to>
</conn>
<conn>
 <from>10</from>
 <to>12</to>
</conn>
<conn>
 <from>11</from>
 <to>12</to>
</conn>

<conn>
 <from>12</from>
 <to>13</to>
</conn>
<conn>
 <from>12</from>
 <to>14</to>
</conn>
<conn>
 <from>12</from>
 <to>15</to>
</conn>

<conn>
 <from>15</from>
 <to>16</to>
</conn>
<conn>
 <from>14</from>
 <to>16</to>
</conn>
<conn>
 <from>13</from>
 <to>16</to>
</conn>
</NEP>
```

Ya que una red de procesadores evolutivos masivamente paralela es equivalente a una red de procesadores evolutivos bajo determinadas restricciones, a continuación se muestran los resultados considerando una red masivamente paralela equivalente.

Listado 4.7: Resultado de la simulación de los 3-colores

```
[----------
Processor 0 : 1
Rules: [[a] --> [XRA, XGA, XBA], [c] --> [YRC, YGC, YBC],
[d] --> [ZRD, ZGD, ZBD], [e] --> [TRE, TGE, TBE]]
Objects: (256) [acde, XRAcde, XGAcde, XBAcde, aYRCde, aYGCde, aYBCde, XRAYRCde,
XRAYGCde, XRAYBCde, XGAYRCde, XGAYGCde, XGAYBCde, XBAYRCde, XBAYGCde,
XBAYBCde, acZRDe, acZGDe, acZBDe, XRAcZRDe, XRAcZGDe, XRAcZBDe, XGAcZRDe,
XGAcZGDe, XGAcZBDe, XBAcZRDe, XBAcZGDe, XBAcZBDe, aYRCZRDe, aYRCZGDe,
aYRCZBDe, aYGCZRDe, aYGCZGDe, aYGCZBDe, acdTRE, aYBCZRDe, acdTGE,
acdTBE, XRAcdTRE, XRAcdTGE, XRAcdTBE, XGAcdTRE, XGAcdTGE,
XGAcdTBE, XBAcdTRE, XBAcdTGE, XBAcdTBE, aYRCdTRE, aYRCdTGE, aYRCdTBE,
aYGCdTRE, aYGCdTGE, aYGCdTBE, aYBCdTRE, aYBCdTGE, aYBCdTBE, XRAYRCdTRE,
XRAYRCdTGE, XRAYRCdTBE, XRAYGCdTRE, XRAYGCdTGE, XRAYGCdTBE, XRAYBCdTRE,
XRAYBCdTGE, XRAYBCdTBE, XGAYRCdTRE, XGAYRCdTGE, XGAYRCdTBE, XGAYGCdTRE,
XGAYGCdTGE, XGAYGCdTBE, XGAYBCdTRE, XGAYBCdTGE, XGAYBCdTBE, XBAYRCdTRE,
XBAYRCdTGE, XBAYRCdTBE, XBAYGCdTRE, XBAYGCdTGE, XBAYGCdTBE, XBAYBCdTRE,
XBAYBCdTGE, XBAYBCdTBE, acZRDTRE, acZRDTGE, acZRDTBE, acZGDTRE, acZGDTGE,
acZGDTBE, acZBDTRE, acZBDTGE, acZBDTBE, XRAcZRDTRE, XRAcZRDTGE, XRAcZRDTBE,
XRAcZGDTRE, XRAcZGDTGE, XRAcZGDTBE, XRAcZBDTRE, XRAcZBDTGE, XRAcZBDTBE,
XGAcZRDTRE, XGAcZRDTGE, XGAcZRDTBE, XGAcZGDTRE, XGAcZGDTGE, XGAcZGDTBE,
XGAcZBDTRE, XGAcZBDTGE, XGAcZBDTBE, XBAcZRDTRE, XBAcZRDTGE, XBAcZRDTBE,
XBAcZGDTRE, XBAcZGDTGE, XBAcZGDTBE, XBAcZBDTRE, XBAcZBDTGE, XBAcZBDTBE,
aYRCZRDTRE, aYRCZRDTGE, aYRCZRDTBE, aYRCZGDTRE, aYRCZGDTGE, aYRCZGDTBE,
aYRCZBDTRE, aYRCZBDTGE, aYRCZBDTBE, aYBCZGDe, aYBCZBDe, XRAYRCZRDe,
XRAYRCZGDe, XRAYRCZBDe, XRAYGCZRDe, XRAYGCZGDe, XRAYGCZBDe, XRAYBCZRDe,
XRAYBCZGDe, XRAYBCZBDe, XGAYRCZRDe, XGAYRCZGDe, XGAYRCZBDe, XGAYGCZRDe,
XGAYGCZGDe, XGAYGCZBDe, XGAYBCZRDe, XGAYBCZGDe, XGAYBCZBDe, XBAYRCZRDe,
XBAYRCZGDe, XBAYRCZBDe, XBAYGCZRDe, XBAYGCZGDe, XBAYGCZBDe, XBAYBCZRDe,
XBAYBCZGDe, XBAYBCZBDe, aYGCZRDTRE, aYGCZRDTGE, aYGCZRDTBE, aYGCZGDTRE,
aYGCZGDTGE, aYGCZGDTBE, aYGCZBDTRE, aYGCZBDTGE, aYGCZBDTBE, aYBCZRDTRE,
aYBCZGDTRE, aYBCZBDTRE, aYBCZRDTGE, aYBCZGDTGE, aYBCZBDTGE, aYBCZRDTBE,
aYBCZGDTBE, aYBCZBDTBE, XRAYRCZRDTRE, XRAYRCZRDTGE, XRAYRCZRDTBE,
XRAYRCZGDTRE, XRAYRCZGDTGE, XRAYRCZGDTBE, XRAYRCZBDTRE, XRAYRCZBDTGE,
XRAYRCZBDTBE, XRAYGCZRDTRE, XRAYGCZRDTGE, XRAYGCZRDTBE, XRAYGCZGDTRE,
XRAYGCZGDTGE, XRAYGCZGDTBE, XRAYGCZBDTRE, XRAYGCZBDTGE, XRAYGCZBDTBE,
XRAYBCZRDTRE, XRAYBCZRDTGE, XRAYBCZRDTBE, XRAYBCZGDTRE, XRAYBCZGDTGE,
XRAYBCZGDTBE, XRAYBCZBDTRE, XGAYRCZRDTRE, XBAYRCZRDTRE, XGAYRCZRDTGE,
XBAYRCZRDTGE, XGAYRCZRDTBE, XBAYRCZRDTBE, XGAYRCZGDTRE, XBAYRCZGDTRE,
XGAYRCZGDTGE, XBAYRCZGDTGE, XGAYRCZGDTBE, XBAYRCZGDTBE, XGAYRCZBDTRE,
XBAYRCZBDTRE, XGAYRCZBDTGE, XBAYRCZBDTGE, XGAYRCZBDTBE, XBAYRCZBDTBE,
XGAYGCZRDTRE, XBAYGCZRDTRE, XGAYGCZRDTGE, XBAYGCZRDTGE, XGAYGCZRDTBE,
XBAYGCZRDTBE, XGAYGCZGDTRE, XBAYGCZGDTRE, XGAYGCZGDTGE, XBAYGCZGDTGE,
XGAYGCZGDTBE, XBAYGCZGDTBE, XGAYGCZBDTRE, XBAYGCZBDTRE, XGAYGCZBDTGE,
XBAYGCZBDTGE, XGAYGCZBDTBE, XBAYGCZBDTBE, XGAYBCZRDTRE, XBAYBCZRDTRE,
XGAYBCZGDTRE, XBAYBCZGDTRE, XGAYBCZBDTRE, XBAYBCZBDTRE, XGAYBCZRDTGE,
```

```
XBAYBCZRDTGE, XGAYBCZGDTGE, XBAYBCZGDTGE, XRAYBCZBDTGE, XGAYBCZBDTGE,
XBAYBCZBDTGE, XGAYBCZRDTBE, XBAYBCZRDTBE, XGAYBCZGDTBE, XBAYBCZGDTBE,
XRAYBCZBDTBE, XGAYBCZBDTBE, XBAYBCZBDTBE]
Output Filter: [X, Y, Z, T]
Input Filter: [Vacio]
----------
, ----------
Processor 1 : 2
Rules: [[XRA] --> [ra], [XGA] --> [ga], [YRC] --> [rc], [YGC] --> [gc]]
Objects: (215) [XRAYRCZRDTRE, XRAYRCZBDTRE, XRAYRCZGDTGE, XRAYRCZRDTBE,
XRAYRCZBDTBE, XRAYGCZGDTRE, XRAYGCZRDTGE, XRAYGCZBDTGE, XRAYGCZGDTBE,
XRAYBCZRDTRE, XRAYBCZBDTRE, XRArcZRDTRE, raYRCZRDTRE, XRAgcZGDTRE,
XRAYBCZGDTGE, XRAYBCZRDTBE, XRAYBCZBDTBE, XGAYRCZGDTRE, XGAYRCZRDTGE,
XGAYRCZBDTGE, XGAYRCZGDTBE, XGAYGCZRDTRE, XGAYGCZBDTRE, XGAYGCZGDTGE,
XGAYGCZRDTBE, XGAYGCZBDTBE, XGAYBCZGDTRE, XGAYBCZRDTGE, XGAYBCZBDTGE,
XGAYBCZGDTBE, XBAYRCZRDTRE, XBAYRCZBDTRE, XBAYRCZGDTGE, XBAYRCZRDTBE,
XBAYRCZBDTBE, XBAYGCZGDTRE, XBAYGCZRDTGE, XBAYGCZBDTGE, XBAYGCZGDTBE,
XBAYBCZRDTRE, XBAYBCZBDTRE, XBAYBCZGDTGE, XBAYBCZRDTBE, XBAYBCZBDTBE,
XRAYRCZRDTGE, XRAYRCZGDTRE, XRAYRCZGDTBE, XRAYRCZBDTGE, XRAYGCZRDTRE,
XRAYGCZRDTBE, XRAYGCZGDTGE, XRAYGCZBDTRE, XRAYGCZBDTBE, XRAYBCZRDTGE,
XRAYBCZGDTRE, XRAYBCZGDTBE, XRAYBCZBDTGE, XGAYRCZRDTRE, XGAYRCZRDTBE,
XGAYRCZGDTGE, XGAYRCZBDTRE, XGAYRCZBDTBE, XGAYGCZRDTGE, XGAYGCZGDTRE,
XGAYGCZGDTBE, XGAYGCZBDTGE, XGAYBCZRDTRE, XGAYBCZRDTBE, XGAYBCZGDTGE,
XGAYBCZBDTRE, XGAYBCZBDTBE, XBAYRCZRDTGE, XBAYRCZGDTRE, XBAYRCZGDTBE,
XBAYGCZRDTRE, XBAYGCZRDTBE, XBAYGCZGDTGE, XBAYGCZBDTRE, XBAYGCZBDTBE,
XBAYBCZRDTGE, XBAYBCZGDTRE, XBAYBCZGDTBE, XBAYBCZBDTGE,
XRArcZBDTRE, XRArcZGDTGE, raYRCZBDTRE, raYRCZGDTGE, raYRCZRDTBE, raYRCZBDTBE,
raYGCZGDTRE, raYGCZRDTGE, raYGCZBDTGE, raYGCZGDTBE, raYBCZRDTRE, raYBCZBDTRE,
rarcZRDTRE, raYBCZGDTGE, raYBCZRDTBE, raYBCZBDTBE, raYRCZRDTGE, raYRCZGDTRE,
raYRCZGDTBE, raYRCZBDTGE, raYGCZRDTRE, raYGCZRDTBE, raYGCZGDTGE, raYGCZBDTRE,
raYGCZBDTBE, raYBCZRDTGE, raYBCZGDTRE, raYBCZGDTBE, raYBCZBDTGE, rarcZBDTRE,
rarcZGDTGE, XRArcZRDTBE, XRAgcZRDTGE, XRAgcZBDTGE, XRAgcZGDTBE, XGAgcZRDTRE,
XGAgcZBDTRE, XGAgcZGDTGE, XGAgcZRDTBE, XGAgcZBDTBE, XBAgcZGDTRE, XBAgcZRDTGE,
XBAgcZBDTGE, XBAgcZGDTBE, XRAgcZRDTRE, XRAgcZRDTBE, XRAgcZGDTGE, XRAgcZBDTRE,
XRAgcZBDTBE, XGAgcZRDTGE, XGAgcZGDTRE, XGAgcZGDTBE, XGAgcZBDTGE, XBAgcZRDTRE,
XBAgcZRDTBE, XBAgcZGDTGE, XBAgcZBDTRE, XBAgcZBDTBE, gaYRCZGDTRE, XRArcZBDTBE,
gaYRCZRDTGE, gaYRCZBDTGE, gaYRCZGDTBE, gaYGCZRDTRE, gaYGCZBDTRE, gaYGCZGDTGE,
gaYGCZRDTBE, gaYGCZBDTBE, gaYBCZGDTRE, gaYBCZRDTGE, gaYBCZBDTGE, gaYBCZGDTBE,
gaYRCZRDTRE, gaYRCZRDTBE, gaYRCZGDTGE, gaYRCZBDTRE, gaYRCZBDTBE, gaYGCZRDTGE,
gaYGCZGDTRE, gaYGCZGDTBE, gaYGCZBDTGE, gaYBCZRDTRE, gaYBCZRDTBE, gaYBCZGDTGE,
gaYBCZBDTRE, gaYBCZBDTBE, gagcZRDTRE, gagcZBDTRE, gagcZGDTGE, gagcZRDTBE,
gagcZBDTBE, gagcZRDTGE, gagcZGDTRE, gagcZGDTBE, gagcZBDTGE, XGArcZGDTRE,
XGArcZRDTGE, XGArcZBDTGE, XGArcZGDTBE, XBArcZRDTRE, XBArcZBDTRE, XBArcZGDTGE,
XBArcZRDTBE, XBArcZBDTBE, XRArcZRDTGE, XRArcZGDTRE, XRArcZGDTBE, XRArcZBDTGE,
XGArcZRDTRE, XGArcZRDTBE, XGArcZGDTGE, XGArcZBDTRE, XGArcZBDTBE, XBArcZRDTGE,
XBArcZGDTRE, XBArcZGDTBE, XBArcZBDTGE, rarcZRDTBE, rarcZBDTBE, rarcZRDTGE,
rarcZGDTRE, rarcZGDTBE, rarcZBDTGE, ragcZRDTGE, ragcZBDTGE, ragcZGDTBE,
ragcZRDTRE, ragcZRDTBE, ragcZGDTGE, ragcZBDTRE, ragcZBDTBE]
Output Filter: [g, r]
Input Filter: [A]
----------
, ----------
Processor 2 : 3
Rules: [[XGA] --> [ga], [XBA] --> [ba], [YGC] --> [gc], [YBC] --> [bc]]
```

```
Objects: (207) [XRAYRCZRDTRE, XRAYRCZBDTRE, XRAYRCZGDTGE, XRAYRCZRDTBE,
XRAYRCZBDTBE, XRAYGCZGDTRE, XRAYGCZRDTGE, XRAYGCZBDTGE, XRAYGCZGDTBE,
XRAYBCZRDTRE, XRAYBCZBDTRE, XRAYBCZGDTGE, XRAYBCZRDTBE, XRAYBCZBDTBE,
XGAYRCZGDTRE, XGAYRCZRDTGE, XGAYRCZBDTGE, XGAYRCZGDTBE, XGAYGCZRDTRE,
XGAYGCZBDTRE, XGAYGCZGDTGE, XGAYGCZRDTBE, XGAYGCZBDTBE, XGAYBCZGDTRE,
XGAYBCZRDTGE, XGAYBCZBDTGE, XGAYBCZGDTBE, XBAYRCZRDTRE, XBAYRCZBDTRE,
XBAYRCZGDTGE, XBAYRCZRDTBE, XBAYRCZBDTBE, XBAYGCZGDTRE, XBAYGCZRDTGE,
XBAYGCZBDTGE, XBAYGCZGDTBE, XBAYBCZRDTRE, XBAYBCZBDTRE, XBAYBCZGDTGE,
XBAYBCZRDTBE, XBAYBCZBDTBE, XRAYRCZGDTRE, XRAYRCZBDTGE, XRAYGCZRDTRE,
XRAYGCZGDTGE, XRAYGCZBDTBE, XRAYBCZRDTGE, XRAYBCZGDTBE, XGAYRCZBDTRE,
XGAYRCZRDTBE, XGAYGCZGDTRE, XGAYGCZBDTGE, XGAYBCZRDTRE, XGAYBCZGDTGE,
XGAYBCZBDTBE, XBAYRCZRDTGE, XBAYRCZGDTBE, XBAYGCZBDTRE, XBAYGCZRDTBE,
XBAYBCZGDTRE, XBAYBCZBDTGE, XRAYRCZRDTGE, XRAYGCZBDTRE, XRAYBCZGDTRE,
XGAYRCZRDTRE, XGAYRCZBDTBE, XGAYGCZGDTBE, XGAYBCZRDTBE, XBAYRCZBDTGE,
XBAYGCZGDTGE, XBAYBCZRDTGE, XRAYRCZGDTBE, XRAYBCZBDTGE, XGAYGCZRDTGE,
XBAYRCZGDTRE, XBAYGCZBDTBE, XRAYGCZRDTBE, XGAYBCZBDTRE, XBAYBCZGDTBE,
XGAYRCZGDTGE, XBAYGCZRDTRE, XRAgcZGDTRE, baYRCZRDTRE, gaYRCZGDTRE,
XRAgcZRDTGE, baYRCZBDTRE, gaYRCZRDTGE, XRAgcZBDTGE, XRAgcZGDTBE, XGAgcZRDTRE,
XGAgcZBDTRE, XRAbcZRDTRE, XRAbcZBDTRE, XRAbcZGDTGE, XRAbcZRDTBE, XRAbcZBDTBE,
XGAbcZGDTRE, XGAbcZRDTGE, XGAbcZBDTGE, XGAbcZGDTBE, XBAbcZRDTRE, XBAbcZBDTRE,
XBAbcZGDTGE, XBAbcZRDTBE, XBAbcZBDTBE, XRAbcZRDTGE, XRAbcZGDTBE, XGAbcZRDTRE,
XGAbcZGDTGE, XGAbcZBDTBE, XBAbcZGDTRE, XBAbcZBDTGE, XRAbcZGDTRE, XGAbcZRDTBE,
XBAbcZRDTGE, XRAbcZBDTGE, XGAbcZBDTRE, XBAbcZGDTBE, XGAgcZGDTGE, XGAgcZRDTBE,
XGAgcZBDTBE, XBAgcZGDTRE, XBAgcZRDTGE, XBAgcZBDTGE, XBAgcZGDTBE, XRAgcZRDTRE,
XRAgcZGDTGE, XRAgcZBDTBE, XGAgcZGDTRE, XGAgcZBDTGE, XBAgcZBDTRE, XBAgcZRDTBE,
XRAgcZBDTRE, XGAgcZGDTBE, XBAgcZGDTGE, XGAgcZRDTGE, XBAgcZBDTBE, XRAgcZRDTBE,
XBAgcZRDTRE, baYRCZGDTGE, baYRCZRDTBE, baYRCZBDTBE, baYGCZGDTRE, baYGCZRDTGE,
baYGCZBDTGE, baYGCZGDTBE, baYBCZRDTRE, baYBCZBDTRE, baYBCZGDTGE, baYBCZRDTBE,
baYBCZBDTBE, baYRCZRDTGE, baYRCZGDTBE, baYGCZBDTRE, baYGCZRDTBE, baYBCZGDTRE,
baYBCZBDTGE, baYRCZBDTGE, baYGCZGDTGE, baYBCZRDTGE, baYRCZGDTRE, baYGCZBDTBE,
baYBCZGDTBE, baYGCZRDTRE, babcZRDTRE, babcZBDTRE, babcZGDTGE, babcZRDTBE,
babcZBDTBE, babcZGDTRE, babcZBDTGE, babcZRDTGE, babcZGDTBE, gaYRCZBDTGE,
gaYRCZGDTBE, gaYGCZRDTRE, gaYGCZBDTRE, gaYGCZGDTGE, gaYGCZRDTBE, gaYGCZBDTBE,
gaYBCZGDTRE, gaYBCZRDTGE, gaYBCZBDTGE, gaYBCZGDTBE, gaYRCZBDTRE, gaYRCZRDTBE,
gaYGCZGDTRE, gaYGCZBDTGE, gaYBCZRDTRE, gaYBCZGDTGE, gaYBCZBDTBE, gaYRCZRDTRE,
gaYRCZBDTBE, gaYGCZGDTBE, gaYBCZRDTBE, gaYGCZRDTGE, gaYBCZBDTRE, gaYRCZGDTGE,
gagcZRDTRE, gagcZBDTRE, gagcZGDTGE, gagcZRDTBE, gagcZBDTBE, gagcZGDTRE,
gagcZBDTGE, gagcZGDTBE, gagcZRDTGE]
Output Filter: [b, g]
Input Filter: [A]
----------
, ----------
Processor 2 : 4
Rules: [[XRA] --> [ra], [XBA] --> [ba], [YRC] --> [rc], [YBC] --> [bc]]
Objects: (216) [XRAYRCZRDTRE, XRAYRCZBDTRE, XRAYRCZGDTGE, XRAYRCZRDTBE,
XRAYRCZBDTBE, XRAYGCZGDTRE, XRAYGCZRDTGE, XRAYGCZBDTGE, XRAYGCZGDTBE,
XRAYBCZRDTRE, XRAYBCZBDTRE, XRAYBCZGDTGE, XRAYBCZRDTBE, XRAYBCZBDTBE,
XGAYRCZGDTRE, XGAYRCZRDTGE, XGAYRCZBDTGE, XGAYRCZGDTBE, XGAYGCZRDTRE,
XGAYGCZBDTRE, XGAYGCZGDTGE, XGAYGCZRDTBE, XGAYGCZBDTBE, XGAYBCZGDTRE,
XGAYBCZRDTGE, XGAYBCZBDTGE, XGAYBCZGDTBE, XBAYRCZRDTRE, XBAYRCZBDTRE,
XBAYRCZGDTGE, XBAYRCZRDTBE, XBAYRCZBDTBE, XBAYGCZGDTRE, XBAYGCZRDTGE,
XBAYGCZBDTGE, XBAYGCZGDTBE, XBAYBCZRDTRE, XBAYBCZBDTRE, XBAYBCZGDTGE,
XBAYBCZRDTBE, XBAYBCZBDTBE, XRAYRCZGDTRE, XRAYRCZBDTGE, XRAYGCZRDTRE,
XRAYGCZGDTGE, XRAYGCZBDTBE, XRAYBCZRDTGE, XRAYBCZGDTBE, XGAYRCZBDTRE,
```

```
XGAYRCZRDTBE, XGAYGCZGDTRE, XGAYGCZBDTGE, XGAYBCZRDTRE, XGAYBCZGDTGE,
XGAYBCZBDTBE, XBAYRCZRDTGE, XBAYRCZGDTBE, XBAYGCZBDTRE, XBAYGCZRDTBE,
XBAYBCZGDTRE, XBAYBCZBDTGE, XRAYRCZRDTGE, XRAYGCZBDTRE, XRAYBCZGDTRE,
XGAYRCZRDTRE, XGAYRCZBDTBE, XGAYGCZGDTBE, XGAYBCZRDTBE, XBAYRCZBDTGE,
XBAYGCZGDTGE, XBAYBCZRDTGE, XRAYRCZGDTBE, XRAYBCZBDTGE, XGAYGCZRDTGE,
XBAYRCZGDTRE, XBAYGCZBDTBE, XRAYGCZRDTBE, XGAYBCZBDTRE, XBAYBCZGDTBE,
XGAYRCZGDTGE, XBAYGCZRDTRE, XRArcZRDTRE, baYRCZRDTRE, XRAbcZRDTRE,
raYRCZRDTRE, raYRCZBDTRE, raYRCZGDTGE, raYRCZRDTBE, raYRCZBDTBE, raYGCZGDTRE,
raYGCZRDTGE, raYGCZBDTGE, raYGCZGDTBE, raYBCZRDTRE, raYBCZBDTRE, raYBCZGDTGE,
raYBCZRDTBE, raYBCZBDTBE, raYRCZGDTRE, raYRCZBDTGE, raYGCZRDTRE, raYGCZGDTGE,
raYGCZBDTBE, raYBCZRDTGE, raYBCZGDTBE, raYRCZRDTGE, raYGCZBDTRE, raYBCZGDTRE,
raYRCZGDTBE, raYBCZBDTGE, raYGCZRDTBE, rarcZRDTRE, baYRCZBDTRE, baYRCZGDTGE,
baYRCZRDTBE, baYRCZBDTBE, baYGCZGDTRE, baYGCZRDTGE, baYGCZBDTGE, baYGCZGDTBE,
baYBCZRDTRE, baYBCZBDTRE, baYBCZGDTGE, baYBCZRDTBE, baYBCZBDTBE, baYRCZRDTGE,
baYRCZGDTBE, baYGCZBDTRE, baYGCZRDTBE, baYBCZGDTRE, baYBCZBDTGE, baYRCZBDTGE,
baYGCZGDTGE, baYBCZRDTGE, baYRCZGDTRE, baYGCZBDTBE, baYBCZGDTBE, baYGCZRDTRE,
XRArcZBDTRE, XRArcZGDTGE, XRArcZRDTBE, XRArcZBDTBE, XRAbcZBDTRE, XRAbcZGDTGE,
XRAbcZRDTBE, XRAbcZBDTBE, XGAbcZGDTRE, XGAbcZRDTGE, XGAbcZBDTGE, XGAbcZGDTBE,
XBAbcZRDTRE, XBAbcZBDTRE, XBAbcZGDTGE, XBAbcZRDTBE, XBAbcZBDTBE, XRAbcZRDTGE,
XRAbcZGDTBE, XGAbcZRDTRE, XGAbcZGDTGE, XGAbcZBDTBE, XBAbcZGDTRE, XBAbcZBDTGE,
XRAbcZGDTRE, XGAbcZRDTBE, XBAbcZRDTGE, XRAbcZBDTGE, XGAbcZBDTRE, XBAbcZGDTBE,
babcZRDTRE, babcZBDTRE, babcZGDTGE, babcZRDTBE, babcZBDTBE, babcZGDTRE,
babcZBDTGE, babcZRDTGE, babcZGDTBE, XGArcZGDTRE, XGArcZRDTGE, XGArcZBDTGE,
XGArcZGDTBE, XBArcZRDTRE, XBArcZBDTRE, XBArcZGDTGE, XBArcZRDTBE, XBArcZBDTBE,
XRArcZGDTRE, XRArcZBDTGE, XGArcZBDTRE, XGArcZRDTBE, XBArcZRDTGE, XBArcZGDTBE,
XRArcZRDTGE, XGArcZRDTRE, XGArcZBDTBE, XBArcZBDTGE, XRArcZGDTBE, XBArcZGDTRE,
XGArcZGDTGE, rarcZBDTRE, rarcZGDTGE, rarcZRDTBE, rarcZBDTBE, rarcZGDTRE,
rarcZBDTGE, rarcZRDTGE, rarcZGDTBE, barcZRDTRE, barcZBDTRE, barcZGDTGE,
barcZRDTBE, barcZBDTBE, barcZRDTGE, barcZGDTBE, barcZBDTGE, barcZGDTRE]
Output Filter: [r, b]
Input Filter: [A]
----------
, ----------
Processor 4 : 5
Rules: []
Objects: (61) [gabcZBDbe, gabcZRDbe, garcZRDbe, garcZGDre, gabcZRDre,
gabcZGDre, gabcgdbe, gabcrdbe, gabcgdre, garcbdre, garcrdbe, garcrdre,
ragcZBDbe, ragcZRDbe, rabcZRDbe, ragcgdbe, ragcbdge, rabcgdbe,
ragcrdbe, ragcbdbe, ragcZBDge, rabcZRDge, rabcZGDge, ragcrdge,
rabcZBDbe, barcZGDge, bagcZRDre, rabcZGDTGE, rabcZBDTBE,
rabcZGDTBE, rabcZBDTGE, gabcZGDbe, garcZBDbe, garcZRDre, garcgdre,
garcgdbe, gabcbdbe, ragcZGDbe, rabcrdge, rabcrdbe, rabcbdbe, rabcbdge,
ragcgdge, bagcZBDge, rabcZRDTBE, rabcZGDTRE, garcZGDbe, gabcZBDre,
garcbdbe, rabcZGDbe, rabcZBDge, ragcZGDge, barcZBDge, rabcZRDTGE,
garcZBDre, gabcbdre, ragcZRDge, rabcZBDTRE, gabcrdre, rabcgdge, rabcZRDTRE]
Output Filter: []
Input Filter: [a, c]
----------
, ----------
Processor 5 : 6
Rules: [[TBE] --> [be], [TGE] --> [ge]]
Objects: (42) [rabcZRDTRE, rabcZBDTBE, rabcZBDTGE, rabcZBDTRE,
rabcZRDTGE, ragcZGDTRE, ragcZRDTRE, ragcZBDTBE, ragcZGDTBE,
ragcZBDTRE, rabcZGDTGE, rabcZGDTRE, ragcZGDTGE, ragcZRDTBE,
```

```
rabcZGDTBE, ragcZBDTGE, ragcZRDTGE, rabcZRDTBE, ragcZBDbe,
ragcZGDge, ragcZBDge, ragcZRDge, rabcZBDge, rabcZRDge, rabcZGDge,
ragcZGDbe, ragcZRDbe, rabcZGDbe, rabcZRDbe, rabcZBDbe, rabcrdbe,
rabcrdge, ragcgdbe, rabcgdge, ragcrdbe, ragcbdge, ragcbdbe, rabcgdbe,
rabcbdge, ragcrdge, rabcbdbe, ragcgdge]
Output Filter: [e]
Input Filter: [ra]
----------
, ----------
Processor 6 : 7
Rules: [[TRE] --> [re], [TBE] --> [be]]
Objects: (18) [garcZBDTGE, garcZGDTGE, garcZRDTGE, garcZBDTRE,
gabcZGDTBE, gabcZRDTBE, gabcZRDTRE, garcZGDTBE, garcZRDTRE,
garcZRDTBE, gabcZGDTGE, gabcZGDTRE, gabcZBDTGE, garcZBDTBE,
gabcZRDTGE, gabcZBDTRE, garcZGDTRE, gabcZBDTBE]
Output Filter: [e]
Input Filter: [ga]
----------
, ----------
Processor 7 : 8
Rules: [[TRE] --> [re], [TGE] --> [ge]]
Objects: (42) [barcZRDTBE, barcZBDTGE, barcZGDTGE, barcZGDTBE,
bagcZGDTRE, bagcZBDTRE, bagcZRDTRE, bagcZRDTBE, bagcZGDTBE,
bagcZBDTBE, barcZBDTRE, barcZRDTRE, barcZGDTRE, bagcZBDTGE,
bagcZRDTGE, barcZBDTBE, barcZRDTGE, bagcZGDTGE, bagcZRDge,
barcZBDre, barcZRDge, bagcZGDge, barcZRDre, barcZGDre, bagcZGDre,
bagcZBDre, bagcZRDre, barcZBDge, barcZGDge, bagcZBDge, barcgdre,
barcgdge, barcrdre, bagcgdge, bagcbdre, barcbdre, barcbdge, bagcrdge,
barcrdge, bagcgdre, bagcrdre, bagcbdge]
Output Filter: [e]
Input Filter: [ba]
----------
, ----------
Processor 8 : 9
Rules: [[ZRD] --> [rd], [ZGD] --> [gd], [ZBD] --> [bd]]
Objects: (90) [rabcZBDbe, rabcZBDge, rabcZGDge, ragcZBDge, ragcZGDbe,
rabcZGDbe, bagcZGDre, bagcZRDre, barcZRDre, barcZBDge, bagcZBDge,
barcZRDge, rabcZRDbe, ragcZBDbe, barcZBDre, barcZGDre, bagcZBDre,
bagcZRDge, bagcZGDge, barcZGDge, rabcZRDge, ragcZRDge, ragcZGDge,
ragcZRDbe, gabcZGDbe, garcZBDre, garcZGDbe, garcZBDbe, garcZRDre,
gabcZBDre, garcZGDre, gabcZBDbe, gabcZRDbe, garcZRDbe, gabcZRDre,
gabcZGDre, bArcrDre, rAbcbDbe, rAbcbDge, rAgcbDge, rAgcbDbe, bArcgDre,
rAgcgDbe, bAgcgDre, rAbcgDbe, bAgcrDre, bArcrDge, bAgcrDge, bArcgDge,
bAgcgDge, rAbcgDge, gArcrDre, gAbcrDbe, gArcrDbe, gAbcrDre, rAgcgDge,
gArcbDbe, gArcbDre, gAbcbDbe, gAbcbDre, rabcgdbe, bagcgdre, barcbdge,
bagcbdge, garcgdre, gabcrdre, garcgdbe, garcbdbe, gabcbdbe, gabcbdre,
rabcrdge, rabcgdge, rabcrdbe, rabcbdbe, rabcbdge, ragcgdge, barcgdre,
bagcgdge, barcgdge, gabcgdbe, gabcgdre, ragcgdbe, ragcbdbe, barcbdre,
bagcbdre, garcbdre, ragcbdge, barcrdge, bagcrdge, ragcrdge]
Output Filter: [d]
Input Filter: []
----------
, ----------
Processor 9 : 10
```

```
Rules: [[ba] --> [bA], [ga] --> [gA], [d] --> [D]]
Objects: (32) [barcrdre, rabcrdbe, rabcrdge, ragcrdbe, bArcrdre, barcrDre,
rabcrDbe, rabcrDge, ragcrDbe, bagcrdre, ragcrdge, bagcrdge, barcrdge,
bAgcrdre, bAgcrdge, bArcrdge, bagcrDre, ragcrDge, bagcrDge, barcrDge,
garcrdre, garcrdbe, gArcrdre, gabcrdbe, gabcrdre, gArcrdbe, gAbcrdbe,
gAbcrdre, garcrDre, garcrDbe, gabcrDbe, gabcrDre]
Output Filter: [D, A]
Input Filter: [rd]
----------
, ----------
Processor 10 : 11
Rules: [[ba] --> [bA], [ra] --> [rA], [d] --> [D]]
Objects: (32) [rabcgdge, rabcgdbe, rAbcgdge, barcgdre, barcgdge, ragcgdbe,
bagcgdre, bagcgdge, bArcgdre, bArcgdge, bAgcgdre, bAgcgdge, rabcgDge,
rAbcgdbe, rAgcgdbe, rabcgDbe, barcgDre, barcgDge, ragcgDbe, bagcgDre,
bagcgDge, gabcgdbe, garcgdre, garcgdbe, gabcgdre, ragcgdge, rAgcgdge,
gabcgDbe, garcgDre, garcgDbe, gabcgDre, ragcgDge]
Output Filter: [D, A]
Input Filter: [gd]
----------
, ----------
Processor 11 : 12
Rules: [[ga] --> [gA], [ra] --> [rA], [d] --> [D]]
Objects: (40) [rabcbdbe, rabcbdge, rAbcbdbe, rabcbDbe, barcbdge, barcbdre,
bagcbdge, ragcbdbe, ragcbdge, bagcbdre, rAbcbdge, rAgcbdbe, rAgcbdge,
rabcbDge, barcbDge, barcbDre, bagcbDge, ragcbDbe, ragcbDge, bagcbDre,
garcbdbe, gabcbdbe, gArcbdbe, garcbdre, gabcbdre, garcbDbe, gabcbDbe,
garcbDre, gabcbDre, gAbcbdbe, gArcbdre, gAbcbdre, rAbcbDge, rAgcbDbe,
rAgcbDge, gArcbDbe, gAbcbDbe, gArcbDre, gAbcbDre, rAbcbDbe]
Output Filter: [D, A]
Input Filter: [bd]
----------
, ----------
Processor 12 : 13
Rules: [[A] --> [a]]
Objects: (52) [bArcrDre, rAbcbDbe, rAbcbDge, rAgcbDge, rAgcbDbe, rAbcgDge,
rAbcgDbe, bArcgDge, bAgcgDge, bArcgDre, bAgcgDre, rAgcgDbe, bAgcrDge,
bAgcrDre, bArcrDge, gArcrDbe, gAbcrDre, gAbcrDbe, gArcrDre, rAgcgDge,
gArcbDbe, gArcbDre, gAbcbDbe, gAbcbDre, ragcbdge, ragcbdbe, bagcrdge,
bagcrdre, garcrDre, ragcgDge, garcbDbe, garcbDre, gabcbDbe, gabcbDre,
barcrDre, rabcbDbe, rabcbDge, ragcbDge, ragcbDbe, rabcgDge, rabcgDbe,
barcgDge, bagcgDge, barcgDre, bagcgDre, ragcgDbe, bagcrDge, bagcrDre,
barcrDge, garcrDbe, gabcrDre, gabcrDbe]
Output Filter: [a]
Input Filter: []
----------
, ----------
Processor 13 : 14
Rules: [[bD] --> [bd], [gD] --> [gd]]
Objects: (8) [barcrDre, barcrDge, garcrDre, barcgDge, barcgDre, garcrDbe,
garcbDbe, garcbDre]
Output Filter: [c, d]
Input Filter: [rc]
----------
```

```
, ----------
Processor 14 : 15
Rules: [[bD] --> [bd], [rD] --> [rd]]
Objects: (10) [ragcbDbe, ragcbDge, bagcgDge, bagcgDre, bagcrDge, bagcrDre,
ragcgDbe, ragcgDge, bagcrdre, ragcbdge]
Output Filter: [c, d]
Input Filter: [gc]
----------
, ----------
Processor 15 : 16
Rules: [[gD] --> [gd], [rD] --> [rd]]
Objects: (11) [rabcbDge, rabcbDbe, rabcgDbe, gabcrDre, rabcgDge, gabcrDbe,
gabcbDbe, gabcbDre, rabcgdbe, rabcgdge, gabcrdbe]
Output Filter: [c, d]
Input Filter: [bc]
----------
, ----------
Processor 16 : 17
Rules: []
Objects: (12) [ragcbdbe, ragcbdge, gabcrdre, rabcgdbe, gabcrdbe, rabcgdge,
bagcrdre, bagcrdge, barcgdge, barcgdre, garcbdre, garcbdbe]
Output Filter: [OO]
Input Filter: []
----------
]
```

Este apartado nos ha mostrado la posibilidad de realizar simulaciones del modelo de Redes de Procesadores Evolutivos, y sus variantes descritas a lo largo de esta Tesis, como modelos capaces de resolver problemas NP en tiempo lineal [57]. Se ha descrito la implementación de tales modelos en un computador tradicional. Para simular el comportamiento no determinista y altamente paralelo se emplean los hilos (threads) que proporciona Java accediendo al multiconjunto de objetos que contienen los procesadores, como consecuencia de la gestión de hilos que realiza la máquina virtual de Java se asegura el no determinismo. La arquitectura de la red de procesadores evolutivos se define empleando el lenguaje de marcas XML, iniciando diferentes hilos de ejecución para los filtros, las reglas, los procesadores, los canales de comunicación y los multiconjuntos de objetos.

La arquitectura empleada por el simulador es una representación genérica del comportamiento de las Redes de Procesadores Evolutivos [55]. Obviamente, la eficiencia teórica nunca se alcanzará debido a la secuencialidad inherente en los computadores tradicionales, pero se pueden reducir los tiempos empleando una implementación distribuida del simulador descrito con

anterioridad en este capítulo.

Las redes de procesadores evolutivos se pueden aplicar al campo de los Sistemas de Ayuda a la Decisión con un enfoque práctico [57], dado el inherente comportamiento basado en reglas de los NEP. Como futuro trabajo de investigación se podrían incorporar los métodos de aprendizaje de las Redes de Neuronas Artificiales adaptándolos al caso simbólico.

Parte V

Conclusiones y Líneas Futuras

Capítulo 5

Conclusiones

En este trabajo se ha realizado un estudio sobre un modelo inspirado en la biología celular denominado redes de procesadores evolutivos [55, 53], esto es, redes cuyos nodos son procesadores muy simples capaces de realizar únicamente un tipo de mutación puntual (inserción, borrado o sustitución de un símbolo). Estos nodos están asociados con un filtro que está definido por alguna condición de contexto aleatorio o de pertenencia. Las redes están formadas a lo sumo de seis nodos y teniendo los filtros definidos por una pertenencia a lenguajes regulares son capaces de generar todos los lenguajes enumerables recursivos independientemente del grafo subyacente. Este resultado no es sorprendente ya que semejantes resultados han sido documentados en la literatura. Si se consideran redes con nodos y filtros definidos por contextos aleatorios –que parecen estar más cerca a las implementaciones biológicas–, entonces se pueden generar lenguajes más complejos, como lenguajes no independientes del contexto. Sin embargo, estos mecanismos tan simples son capaces de resolver problemas complejos en tiempo polinomial. Se ha presentado una solución lineal para un problema NP-completo, el problema de los 3-colores.

Como primer aporte significativo se ha propuesto una nueva dinámica de las redes de procesadores evolutivos con un comportamiento no determinista y masivamente paralelo [55], y por tanto todo el trabajo de investigación en el

área de la redes de procesadores se puede trasladar a las redes masivamente paralelas. Por ejemplo, las redes masivamente paralelas se pueden modificar de acuerdo a determinadas reglas para mover los filtros hacia las conexiones. Cada conexión se ve como un canal bidireccional de manera que los filtros de entrada y salida coinciden. A pesar de esto, estas redes son computacionalmente completas. Se pueden también implementar otro tipo de reglas para extender este modelo computacional. Se reemplazan las mutaciones puntuales asociadas a cada nodo por la operación de splicing. Este nuevo tipo de procesador se denomina procesador splicing. Este modelo computacional ANSP es semejante en cierto modo a los sistemas distribuidos en tubos de ensayo basados en splicing.

Además, se ha definido un nuevo modelo [56] –Redes de procesadores evolutivos con filtros en las conexiones–, en el cual los procesadores tan sólo tienen reglas y los filtros se han trasladado a las conexiones. Dicho modelo es equivalente, bajo determinadas circunstancias, a las redes de procesadores evolutivos clásicas. Sin dichas restricciones el modelo propuesto es un superconjunto de los NEPs clásicos. La principal ventaja de mover los filtros a las conexiones rádica en la simplicidad de la modelización.

Otra de las aportaciones de este trabajo ha sido el diseño de un simulador en Java [54, 52] para las redes de procesadores evolutivos propuestas en esta Tesis.

Sobre el término "procesador evolutivo" empleado en esta Tesis, el proceso computacional descrito aquí no es exactamente un proceso evolutivo en el sentido Darwiniano. Pero las operaciones de reescritura que se han considerado pueden interpretarse como mutaciones y los procesos de filtrado se podrían ver como procesos de selección. Además, este trabajo no abarca la posible implementación biológica de estas redes, a pesar de ser de gran importancia.

Todas estas aportaciones se pueden ver reflejadas en las publicaciones del autor.

A lo largo de esta tesis se ha tomado como definición de la medida de complejidad para los ANSP, una que denotaremos como tamaño (considerando

tamaño como el número de nodos del grafo subyacente). Se ha mostrado que cualquier lenguaje enumerable recursivo L puede ser aceptado por un ANSP en el cual el número de procesadores está linealmente acotado por la cardinalidad del alfabeto de la cinta de un máquina de Turing que reconoce dicho lenguaje L. Siguiendo el concepto de ANSP universales introducido por Manea [65], se ha demostrado que un ANSP con una estructura de grafo fija puede aceptar cualquier lenguaje enumerable recursivo. Un ANSP se puede considerar como un ente capaz de resolver problemas, además de tener otra propiedad relevante desde el punto de vista práctico: Se puede definir un ANSP universal como una subred, donde sólo un cantidad limitada de parámetros es dependiente del lenguaje. La anterior característica se puede interpretar como un método para resolver cualquier problema NP en tiempo polinomial empleando un ANSP de tamaño constante, concretamente treinta y uno. Esto significa que la solución del cualquier problema NP es uniforme en el sentido de que la red, exceptuando la subred universal, se puede ver como un programa: adaptándolo a la instancia del problema a resolver, se escogerán los filtros y las reglas que no pertenecen a la subred universal. Un problema interesante desde nuestro punto de vista es el que hace referencia a como elegir el tamaño óptimo de esta red

Capítulo 6

Líneas Futuras

Una vez concluido el trabajo de esta tesis se abre un abanico de posibilidades y preguntas sin contestar que pudieran ser posibles continuaciones de este trabajo de investigación. Se intentara enumerar y resumir las mas relevantes:

6.1. Filtros en las conexiones

En [56] se demuestra que un ANSP es computacionalmente completo. De lo que cabe preguntar si un ASNPFC será computacionalmente completo. Si éste no es el caso, ¿qué tipo de problemas se pueden resolver eficientemente empleando el modelo descrito?. Además, el tipo de problemas NP es exactamente igual al tipo de todos los lenguajes decidibles por un ANSP en tiempo polinomial. Después de lo anteriormente expuesto surge la siguiente interrogante: ¿Se pueden caracterizar los ANSPFC del mismo modo que los ANSP?.

6.2. Universalidad

Partiendo de la premisa de que los datos pueden transformarse en palabras, surge el concepto de redes de procesadores paralelos de lenguajes con el objetivo de aplicarlo a la teoría de lenguajes y gramáticas formales. Cada procesador ubicado en los nodos de la red es un procesador simple, denominado procesador evolutivo. Hay que tener en cuenta que éste no es un objeto real sino un concepto matemático. Por Procesador Evolutivo se entiende aquel que es capaz de realizar operaciones muy simples, es decir, operaciones sobre lenguajes formales que enmascaran mutaciones puntuales en secuencias de ADN (inserciones, borrados o sustituciones de un par de nucleótidos). Cada nodo se puede considerar como una célula con información genética codificada en secuencias de ADN que pueden evolucionar por eventos locales, mutaciones. Cada nodo se especializa en una de estas operaciones. Los datos dentro de cada nodo se organizan en la forma de multiconjuntos de palabras y todas las copias se procesan en paralelo de forma que todos los posibles eventos que puedan tener lugar se efectuen. Desde el punto de vista biológico no se espera que los componentes de un organismo evolucionen secuencialmente o que la reproducción de las células se modele con una alternativa secuencial. Los cambios en el estado de las células se modelizan con reglas de reescritura como en los lenguajes formales. La naturaleza paralela en los cambios de estado de las células se modelizan empleando la ejecución paralela de la reescritura de símbolos dependiendo de las reglas que se apliquen. Consecuentemente los HNEP se pueden considera modelos de computación bioinspirados. La recombinación no está presente pero se puede afirmar que las relaciones funcionales y evolutivas entre los genes se pueden representar tomando mutaciones locales.

Los mecanismos descritos se están abordando en una serie de trabajos posteriores [13, 12, 67], que abordan el tema de dispositivos generadores de lenguajes y su poder computacional. Por otro lado, este modelo se considera un dispositivo que acepta lenguajes, donde se ha obtenido una nueva caracterización de NP. Los modelos comentados, además de la motivación matemática, pueden tener también su vertiente biológica. Las células siempre forman tejidos y órganos que interactúan con otros directamente o a través

del entorno común.

6.3. Picture NEPS

Trasladar esta investigación a los lenguajes Picture , estudiados ámpliamente en la literatura existente al respecto, se han definidos empleando diferentes mecanismos. Modelos arrays y matrices de dos dimensiones de los Pictures mencionados han sido propuestos como arrays rectangulares de símbolos en [101, 94, 91, 90]. También estos modelos de definición de Pictures han sido tratados como Arrays no necesariamente rectangulares en los años 70 [88] y una jerarquía de estas gramáticas fue expuesta en [100, 95]. Otro tipo de clases de gramáticas para la generación de Picture , no necesariamente rectangulares fue definida en [73]. Un nuevo modelo de reconocedores de lenguajes Picture, extendiendo a dos dimensiones la caracterización de los lenguajes de Picture en términos de un alfabeto MORPHINS de lenguajes locales, ha sido introducido en [31]. Una reciente recopilación de autómatas reconocedores de lenguajes Picture [37], otra recopilación que considera diferentes mecanismos para definir lenguajes Picture, no necesariamente rectangulares, aparece referenciada en [88, 32].

6.4. Implementación Hardware

Formalización y diseño de un sistema hardware que soporte a los procesadores evolutivos, teniendo en cuenta en el diseño que estos procesadores describen elementos y manejan componentes muy cercanos a la biología. Esta modelización en sistemas físicos de los Procesadores Evolutivos en una tarea compleja que requiere la división del sistema completo en varios circuitos con objetivos particulares más concretos. Además, el procedimiento general para obtener un sistema hardware debe recorrer una secuencia ordenada de sucesivas etapas o fases. Primeramente habría que diseñar los circuitos que permiten obtener y aplicar las reglas de evolución para un Procesador Evolutivo. Además, se debe realizar la síntesis del circuito que se ocupe de seleccionar

las reglas a aplicar para que un Procesador evolucione desde un estado al siguiente. Este circuito dará como salida aquellas reglas que cumplan todas las condiciones necesarias para ser aplicadas en un paso de evolución del sistema: 1. Definición del modelo formal que permita caracterizar e implementar un Procesador Evolutivo.

2. Definición del esquema de las estructuras de datos.

3. Obtención de los componentes hardware para almacenar los datos.

4. Descripción de cada elemento o bloque funcional.

5. Diseño de un modelo de descripción de hardware.

6. Implementación y prueba del circuito diseñado.

6.5. Aprendizaje

A la hora de incorporar aprendizaje en las redes de procesadores evolutivos se puede realizar un estudio sobre las semejanzas de las Redes de Neuronas Artificiales y las Redes de Procesadores Evolutivos, para incluir una etapa de aprendizaje en éstas últimas siendo capaces de resolver diferentes problemas con la misma arquitectura. La gran desventaja de una red tradicional es que únicamente pueden resolver un problema dado, si es necesario resolver un problema similar se necesitará otra red diferente. La idea del aprendizaje trata de sobrepasar esta desventaja proponiendo un modelo capaz de resolver diferentes clases de problemas. Evidentemente, existen los NEP universales que sí son capaces de resolver cualquier problema, pero habría que realizar un estudio sobre la capacidad de recursos y la complejidad computacional de éstos frente a una red con aprendizaje [74, 43].

Para la incorporación de un modelo de aprendizaje en los filtros ubicados en las conexiones se podría definir un algoritmo de aprendizaje simbólico basado en los Mapas Autoorganizados [1], ya que toda la información que se maneja son cadenas de símbolos. Cabría describir tres modelos de aprendizaje que se pueden incorporar en el funcionamiento de las redes de procesadores

evolutivos. El primero se basaría en el aprendizaje de los filtros de manera que las cadenas que los atraviesan (o no) modifican dichos filtros. El segundo se basaría en que las cadenas que se comunican a través de las conexiones se ven modificadas de acuerdo a un aprendizaje marcado por los filtros, pero éstos no son modificados. Tal aprendizaje solventa la secuencialidad del aprendizaje sobre los filtros y permite la misma paralelización de que disponen las redes de procesadores evolutivos. Y por último una combinación de ambos.

Parte VI

Bibliografía

Bibliografía

[1] *A. J. Abrantes and J. S. Marques. Exploiting the common structure of SOM edge linking algorithms: an experimental study. In* Proceedings of the International Conference on Image Processing*, volume 3, pages 624–7. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.*

[2] *Leonard Addleman. Molecular computation of solutions to combinatorial problems.* Science*, page 1021, 1994.*

[3] *Leonard Addleman. On the path to computation with dna.* Science*, (993), 1994.*

[4] *Artiom Alhazov, Carlos Martín-Vide, and Linqiang Pan. Solving a PSPACE-complete problem by recognizing P systems with restricted active membranes.* Fundamenta Informaticae*, 58(2):67–77, 2003.*

[5] *Artiom Alhazov and Linqiang Pan. Polarizationless P systems with active membranes.* Grammars*, 7:141–159, 2004.*

[6] *Artiom Alhazov, Linqiang Pan, and Gheorghe Păun. Trading polarizations for labels in P systems with active membranes.* Acta Informatica*, 41(2-3):111–144, December 2004.*

[7] *F. Bernardini and M. Gheorghe. Cell communication in tissue p systems: universality results.* Soft Computing*, 9(9):640–649, September 2005.*

[8] *Daniela Besozzi, Giancarlo Mauri, Gheorghe Păun, and Claudio Zandron. Gemmating P systems: collapsing hierarchies.* Theoretical Computer Science*, 296(2):253–267, March 2003.*

[9] *Daniela Besozzi, Claudio Zandron, Giancarlo Mauri, and N. Sabadini. P systems with gemmation of mobile membranes. In Antonio Restivo, Simona Ronchi-Della-Rocca, and Luca Roversi, editors,* Theoretical Computer Science. 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001. Proceedings.*, volume 2202 of* Lecture Notes in Computer Science*, pages 136–153, Turin, October 2001. Springer-Verlag.*

[10] *Luca Bianco, Federico Fontana, Giuditta Franco, and Vincenzo Manca. P Systems in bio systems. Submitted, 2004. in G. Paun, P systems: Applications and Perspectives, to appear, 2004.*

[11] *R. Brijder, G. Rozenberg, M. Cavaliere, A. Riscos-Nunez, and D. Sburlan. Communication membrane systems with active symports. Submitted, 2005.*

[12] *J. Castellanos, C. Martín-Vide, V. Mitrana, and J. Sempere. Solving np-complete problems with networks of evolutionary processors.* Lecture Notes in Computer Science*, 2084:621–628, 2001.*

[13] *J. Castellanos, C. Martín-Vide, V. Mitrana, and J. Sempere. Networks of evolutionary processors.* Acta Informática*, 39:517–529, 2003.*

[14] *Juan Castellanos, Florin Manea, Fernando de Mingo, and Victor Mitrana. Accepting networks of splicing processors with filtered connections.* Lecture Notes in Computer Science*, 4664:218–229, 2007.*

[15] *Juan Castellanos, Gheorghe Păun, and Alfonso Rodríguez-Patón. P systems with worm-objects. In* Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 64–74, A Coruña, Spain, September 2000. IEEE Computer Society.*

[16] *Matteo Cavaliere. Evolution, communication and observation. from biology to membrane systems and back. Submitted. RNGC Report 03/2004, Sevilla University.*

[17] *Matteo Cavaliere and Ioan I. Ardelean. Modelling respiration in bacteria and respiration/photosynthesis interaction in cyanobacteria by using a P System simulator. Submitted, 2004.*

[18] *H. Chen, R. Freund, M. Ionescu, Gheorghe Păun, and M.J. Perez-Jimenez. On string languages generated by spiking neural p systems. Submitted, 2006.*

[19] *G. Ciobanu and Gheorghe Păun. The minimal parallelism is still universal (for p systems with symport/antiport rules), 2005.*

[20] *Gabriel Ciobanu. Distributed computing in P Systems with antiport communication. Submitted, 2002.*

[21] *Gabriel Ciobanu, Vlad Ciubotariu, and Bogdan Tanasa. A computational model of membrane transportation. Submitted.*

[22] *A. Cordon-Franco, M.A. Gutierez-Naranjo, M.J. Perez-Jimenez, and A. Riscos-Nunez. Cellular solutions to some numerical np-complete problems. a prolog implementation, molecular computational models. Unconventional Approaches (M. Gheorghe, ed.), 2004. Idea-Group, London 2004, 115–149.*

[23] *Erzsébet Csuhaj-Varjú, Carlos Martín-Vide, Gheorghe Păun, and Arto Salomaa. From Watson-Crick L Systems to Darwinian P systems.* Natural Computing*, 2(3):299–318, August 2003.*

[24] *Erzsébet Csuhaj-Varjú, Gheorghe Păun, and Gyorgy Vaszil. Grammar systems vs. membrane computing: The case of CD grammar systems. Submitted, 2004.*

[25] *Emilio del Rosal, Rafael Nuñez, Carlos Casteñeda, and Alfonso Ortega. Simulating neps in a cluster with jnep.* Computers, Communications and Control*, III:480–485, 2008.*

[26] *Hiroshi Douzono, Shigeomi Hara, and Yoshio Noguchi. Clustering method of chromosome fluorescence profiles using modified self organizing map controlled by simulated annealing. In* Proceedings of the International Joint Conference on Neural Networks*, volume 4, pages 103–106, Piscataway, NJ, 2000. Saga Univ, IEEE.*

[27] *R. Freund and A. Paun. P systems with active membranes and without polarizations.* Soft Computing*, 9(9):657–663, September 2005.*

[28] *Rudolf Freund and Marion Oswald. Tissue p systems with symport/antiport rules of one symbol are computationally universal. Submitted, 2005.*

[29] *Michael R. Garey and David S. Johnson.* Computers and Intractability, A Guide to the Theory of NP-Completeness. *W. H. Freeman and Company, 1979.*

[30] *M. Gheorghe and Gheorghe Păun. Computing by self-assembly: Dna molecules, polyominoes, cells. Submitted, 2005.*

[31] *D. Giammarresi and A. Restivo. Recognizable picture languages.* Pattern Recognition and Artificial Intelligence*, 6:241–256, 1992.*

[32] *D. Giammarresi and A. Restivo.* Handbook of Formal Languages*, chapter Two-dimensional languages, pages 215–267. Springer-Verlag, 1997.*

[33] *C. Graciani and A. Riscos-Nunez. Looking for simple common schemes to design recognizer p systems with active membranes that solve numerical decision problems. UC05, accepted, 2005.*

[34] *Thore Graepel, Matthias Burger, and Klaus Obermayer. Deterministic annealing for topographic vector quantization and self-organizing maps. In* Proceedings of WSOM'97, Workshop on Self-Organizing Maps, Espoo, Finland, June 4–6*, pages 345–350. Helsinki University of Technology, Neural Networks Research Centre, Espoo, Finland, 1997.*

[35] *M. Gutierrez-Naranjo and M.J. Perez-Jimenez. P systems with active membranes, without polarizations and without dissolution: a characterization of p. UC05, accepted, 2005.*

[36] *O.H. Ibarra and Gheorghe Păun. Characterizations of context-sensitive languages and other language classes in terms of symport/antiport p systems. Submitted, 2005.*

[37] *I. Inoue and I. Takanami. A survey of two-dimensional automata theory. In* 5th Int. Meeting of Young Computer Scientists*, volume 381, pages 72–91, 1990.*

[38] *Tseren Onolt Ishdorj and Mihai Ionescu. Replicative-distribution rules in P Systems with active membranes. Submitted, 2004. First International Colloquium on THEORETICAL ASPECTS OF COMPUTING Guiyang, China 20 - 24 September 2004.*

[39] *Sungchul Ji and Gabriel Ciobanu. Towards the modeling of cell communication and computation using the shape algebra of biopolymers. psystems.disco.unimib.it, 2002. psystems.disco.unimib.it.*

[40] *L. Kari, G. Gloor, and S. Yu. Using dna to solve the bounded correspondence problem.* Theoretical Computer Science*, 231:193–203, 2000.*

[41] *Shankara Narayanan Krishna and Raghavan Rama. A variant of P systems with active membranes: Solving NP-Complete problems.* Romanian Journal of Information Science and Technology*, 2(4):357–367, 1999.*

[42] *S. Y. Kuroda. Classes of languages and linear-bounded automata.* Information and Control*, 7(2):207–223, 1964.*

[43] *Mingo L., Arroyo F., Díaz M.A., and Catellanos J. Hierarchical knowledge representation: Symbolic conceptual trees and universal approximation.* International Journal of Intelligent Control and Systems*, 2(2):142–149, 2007.*

[44] *K. Lakshmanan. Computational universality and solving NP complete problems using insertion deletion tissue P Systems. Submitted, 2003.*

[45] *Mingo L.F, Aslanyan L, Catellanos J, Díaz M.A., and Riazanov V. Fourier neural networks: An approach with sinusoidal activation functions.* International Journal on INFORMATION THEORIES & APPLICATIONS*, 11(1):52–55.*

[46] Mingo L.F, Aslanyan L, Catellanos J, Riazanov V., and Díaz M.A. Context Neural Network for Temporal Correlation and Prediction*, pages 110–113. Artificial Neural Nets and Genetic Algorithms, 2001.*

[47] Mingo L.F, Díaz M.A., Gisbert F, and Carrillo J.D. Enhanced neural networks: An overview. Knowledge-based Intelligent Information Knowledge-based Intelligent Information Engineering Systems and Allied Technologies Frontiers in Artificial Intelligence and Applications*, 69 Part II:1170–1173, 2001.*

[48] Mingo L.F, Díaz M.A., Pérez M, and Palencia V. Enhanced neural networks as taylor series approximation. KnowledgeBased Intelligent Information Engineering Systems & Allied Technologies Frontiers in Artificial Intelligence and Applications*, 82:532–533, 2002.*

[49] Mingo L.F, Díaz M.A., Gonzalo R., López G., and Gisbert F. Neural networks with genetic controllers: Trade sector forecasting. WSEAS Transaction on Computers*, 2:274–277, 2003.*

[50] Mingo L.F, Díaz M.A., Gonzalo R., Aslanyan L, and Santos E. Data flows algorithms with neural networks. WSEAS Transaction on Circuits and Systems*, 2:619–624.*

[51] Mingo L.F, Díaz M.A., Palencia V., Santos E., and Jiménez P. IBEX-35 Stock Market Forecasting Using Time Delay Connections in Enhanced Neural Networks*, pages 455–460. World Multiconference on Systemics, Cybernetics and Informatics, 2002.*

[52] Peña M, Diaz M.A., and Mingo L.F. Networks of Evolutionary Processors UML Architecture*, pages 176–183. 5th WSEAS international conference on Computation Intelligence Man, Machine System and Cybernetic, 2006.*

[53] Díaz M.A., Mingo L.F, and Gomez Nuria. Networks of evolutionary processors: Java implementation of a threaded processor. Information Research and Applications*, 1:203–210, 2007.*

[54] *Díaz M.A., Mingo L.F, and Gomez Nuria. Networks of evolutionary processors: Java implementation of a threaded processor.* Information Research and Applications*, 15:37–43, 2008.*

[55] *Díaz M.A., Mingo L.F, Gomez Nuria, and Catellanos J. Implementation of massive parallel networks of evolutionary processors (mpnep): 3-colorability problem.* International Workshop on Nature Inspired Cooperative Strategies for Optimization. Studies in Computational Intelligence, Springer*, pages 8–10, 2008.*

[56] *Díaz M.A., Peña M, and Mingo L.F. Simulation of networks of evolutionary processors with filtered connection.* WSEAS Press*, 4:608–616.*

[57] *Díaz M.A., Gomez Nuria, Santos E., Gonzalo R., and Gisbert F. Networks of evolutionary processors and decision support system.* nternational Conference Information Research and Aplicaction*, 197-202, 2007.*

[58] *Mutyam Madhu. Rewriting P systems. collapsing hierarchies. Submitted. Theoretical Computer Science, to appear.*

[59] *Mutyam Madhu. A note on P Systems with replicated rewriting. Submitted, 2002.*

[60] *Mutyam Madhu and Kamala Krithivasan. Inter-membrane communication in P systems.* Romanian Journal of Information Science and Technology*, 3(4):335–352, 2000.*

[61] *Mutyam Madhu and Kamala Krithivasan. P systems with dynamic membrane polarization.* Romanian Journal of Information Science and Technology*, 4(1-2):135–154, 2001.*

[62] *Mutyam Madhu and Kamala Krithivasan. P systems with membrane creation: Universality and efficiency. In Maurice Margenstern and Yurii Rogozhin, editors,* Machines, Computations, and Universality. Third International Conference, MCU 2001 Chisinau, Moldava, May 23-27, 2001. Proceedings.*, volume 2055 of* Lecture Notes in Computer Science*, pages 276–287, Berlin, 2001. Springer-Verlag.*

[63] *Mutyam Madhu and Kamala Krithivasan. Tissue P Systems with leftmost rewriting. Submitted, 2004.*

[64] *Mutyam Madhu, Vadali S. Murty, and Kamala Krithivasan. Hardware realization of P Systems with carriers. Poster presentation in the Eighth International Conference on DNA based Computers, Hokkaido University, Sapporo Campus, Japan, June 10-13, 2002, June 2002. Poster presentation in the Eighth International Conference on DNA based Computers, Hokkaido University, Sapporo Campus, Japan, June 10-13, 2002.*

[65] *Florin Manea and Victor Mitrana. All np-problems can be solved in polynomial time by accepting hybrid networks of evolutionary processors of constant size.* Information Processing Letters*, 103(3):112–118, 2007.*

[66] *Salomon Marcus. Membranes versus DNA.* Fundamenta Informaticae*, 49(1-3):223–227, January 2002. Special Issue: Membrane Computing (WMC-CdeA2001) Guest Editor(s): Carlos Martín-Vide, Gheorghe Păun.*

[67] *C. Martin-Vide, V. Mitrana, M. Perez-Jimenez, and F. Sancho Caparrini. Hybrid networks of evolutionary processors.* Lecture Notes in Computer Science*, 2723:401–412, 2003.*

[68] *Carlos Martín-Vide, Gheorghe Păun, Juan Pazos, and Alfonso Rodríguez-Patón. Tissue P systems.* Theoretical Computer Science*, 296(2):295–326, March 2003.*

[69] *M. Muskulus and R. Brijder. Complexity of biocomputation: symbolic dynamics in membrane systems.* Intern. J. Found. Computer Sci. *To Appear.*

[70] *Madhu Mutyam and Kamala Krithivasan. P systems with membrane creation: Universality and efficiency. In Y. Rogozhin M. Margenstern, editor,* Machines, Computations, and Universality: Third International Conference, MCU 2001 Chisinau, Moldavia, May 23-27, 2001, Proceedings*, volume 2055 of* Lecture Notes In Computer Science*, pages 276–287. Springer-Verlag Heidelberg, May 23-27 2001.*

[71] *Madhu Mutyam, Vaka Jaya Prakash, and Kamala Krithivasan. Rewriting tissue P systems.* Journal of Universal Computer Science, *10(9):1250–1271, September 2004.*

[72] *Isabel A. Nepomuceno-Chamorro. A Java simulator for membrane computing.* Journal of Universal Computer Science, *10(5):620–629, May 2004.*

[73] *M. Nivat, A. Saoudi, K.G. Subramanian, R. Siromoney, and V.R. Dare. Puzzle grammars and context-free array grammars.* Pattern Recognition and Artificial Inteligence, *(5):663–676, 1991.*

[74] *Gomez Nuria, Santos E., and Díaz M.A. Symbolic learning(clustering) over dna string.* WSEAS Press, *4:617–624, 2007.*

[75] *Adam Obtulowicz. Note on some recursive family of P Systems with active membranes. Submitted, 2001.*

[76] *L. Pan, A. Alhazov, and T.O. Isdorj. Further remarks on p systems with active membranes, separation, merging, and release rules.* Soft Computing, *9(9):686–690, September 2005.*

[77] *A. Paun and B. Popa. P systems with proteins on membranes. Submitted, 2005.*

[78] *A. Paun and B. Popa. Rewriting p systems with communication by symport rules. Submitted, 2006.*

[79] *Andrei Păun, Gheorghe Păun, and Grzegorz Rozenberg. Computing by communication in networks of membranes.* International Journal of Foundations of Computer Science, *13(6):779–798, December 2002.*

[80] *G. Paun, G. Rozenberg, and A. Saloma.* DNA Computing, New Computing Paradigms. *Springer-Verlag, 1998.*

[81] *Gheorghe Păun. Computing with membranes (P systems): A variant.* International Journal of Foundations of Computer Science, *11(1):167–182, March 2000. and CDMTCS TR 098, Univ. of Auckland, 1999 (www.cs.auckland.ac.nz/CDMTCS).*

[82] *Gheorghe Păun.* Membrane Computing. An Introduction. *Springer-Verlag, Berlin, 2002.*

[83] *Gheorghe Păun and Grzegorz Rozenberg. A guide to membrane computing.* Theoretical Computer Science*, 287(1):73–100, September 2002.*

[84] *Mario J. Pérez-Jimenez and Francisco José Romero-Campero. Modelling egfr signalling network using continuous membrane systems. Submitted, 2005.*

[85] *M.J. Perez-Jimenez and F.J. Romero-Campero. Modelling vibrio fischeri's behaviour using p systems. accepted in the Systems Biology Workshop, ECAL 2005, September 2005.*

[86] *A. Rodriguez-Paton and P. Sosik. P systems with active membranes characterize PSPACE. Submitted, 2005.*

[87] *Alfonso Rodriguez-Patón. Computing with membranes: P Systems with DNA-Worms. GECCO, 2001 (poster), 2001. GECCO, 2001 (poster).*

[88] *A. Rosenfeld and R. Siromoney. Picture languages – a survey.* Languages of design*, (1):229–245, 1993.*

[89] *G. Rozenberg and A. Saloma.* The Handbook of Formal Languages*. Springer-Verlag, 1997.*

[90] *G. Siromoney, R. Siromoney, and K. Krithivasan. Abstract families of matrices and picture languages.* Computer Graphics and Image Processing*, (1):284–307, 1972.*

[91] *G. Siromoney, R. Siromoney, and K. Krithivasan. Picture languages with array rewriting rules.* Information and Control*, (22):447–470, 1973.*

[92] *Petr Sosík. The power of catalysts and priorities in membranes. Submitted, 2002.*

[93] *Petr Sosík. The computational power of cell division in P systems: Beating down parallel computers?* Natural Computing*, 2(3):287–298, August 2003.*

[94] *K.G. Subramanian and R. Siromoney. On array grammars and languages.* Cybernetics and Systems*, 18:77–98, 1987.*

[95] *K.G. Subramanian, R. Siromoney, V.R. Dare, and A. Saoudi. Basic puzzle languages.* Pattern Recognition and Artificial Intel ligence,*, (9):763–775, 1995.*

[96] *Yasuhiro Suzuki, Daniela Besozzi, Claudio Zandron, Hiroshi Tanaka, and Giancarlo Mauri. Toward a novel computational framework for molecular computing: chemical reaction as computation. Submitted, 2004. DNA10, Milano, 2004.*

[97] *Head T., Yamamura M., and Gal S. Aqueous computing: writting on molecules. In IEEE Service Center, editor,* Proceedings of the Congress on Evolutionary Computation*, pages 1006–1010. Piscataway NJ, 1999.*

[98] *Sergei Verlan. Communicating distributed h systems with alternating filters and tissue p systems with minimal symport/antiport, 2003. EMCC Workshop - 2nd Annual MolCoNet Meeting November 27-29, 2003 Wien, Austria.*

[99] *Sergey Verlan. Tissue P Systems with minimal symport/antiport, 2004. DLT'04 - Eighth International Conference on Developments in Language Theory, Auckland, New Zealand - December 13-17 2004.*

[100] *P.S. Wang. Hierarchical structure and complexities of parallel isometric patterns.* IEEE Trans. PAM*, 1(5):92, 1975.*

[101] *P.S. Wang. Sequential/parallel matrix array languages.* Journal of Cybernetics*, 5:19–36, 1975.*

Parte VII

Publicaciones del Autor

1. *Mingo L.F., Díaz M.A., Gisbert F., Carrillo J.D.:* **Enhanced Neural Networks: An Overview**. *Knowledge-based Intelligent Information Engineering Systems and Allied Technologies. Frontiers in Artificial Intelligence and Applications. IOS Press Ohmsha. ISSN: 0922-6389. ISSN: 1535-6698. Vol.: 69. Part II. Pp.: 1170-1173. 2001.*

2. *Mingo L.F., Aslanyan L., Castellanos J., Riazanov V., Díaz M.A.:* **Context Neural Network for Temporal Correlation and Prediction**. *Artificial Neural Nets and Genetic Algorithms. Springer Computer Science. ISBN: 3-211-83651-9. Pp.: 110-113. 2001.*

3. *Mingo L.F., Díaz M.A., Palencia V., Santos E., Jiménez P.:* **IBEX-35 Stock Market Forecasting Using Time Delay Connections in Enhanced Neural Networks**. *World Multiconference on Systemics, Cybernetics and Informatics. SCI 2002. July 14-18, Orlando USA. ISBN: 980-07-81-50. Pp.: 455-460. 2002.*

4. *Luis F. Mingo, Miguel A. Díaz, Pérez M., Palencia V.:* **Enhanced Neural Networks as Taylor Series Approximation**. *Knowledge-Based Intelligent Information Engineering Systems & Allied Technologies. Frontiers in Artificial Intelligence and Applications. Vol. 82. IOS Press. ISSN: 0922-6389. Pp.: 532-533. 2002.*

5. *de Mingo L.F., Diaz M., Gonzalo R., Lopez G., Gisbert P.:* **Neural Networks with Genetic Controllers: Trade Sector Forecasting**. *WSEAS Transaction on Computers. Issue 1, Vol. 2. ISSN 1109-2750. Pp: 274-277. 2003*

6. *de Mingo L.F., Diaz M., Gonzalo R., Aslanyan L., Santos E.:* **Data Flows Algorithms with Neural Networks**. *WSEAS Transaction on Circuits and Systems. Issue 3, Vol. 2. ISSN 1109-2734. Pp: 619-624. 2003.*

7. *Luis Mingo, Levon Aslanyan, Juan Castellanos, Miguel Diaz, and Vladimir Riazanov.* **Fourier Neural Networks: An Approach with Sinusoidal Activation Functions**. *International Journal on INFORMATION THEORIES & APPLICATIONS. Vol. 11. No. 1. ISSN 1310-0513. Pp.: 52-55. 2004.*

8. *Luis F. Mingo, Arroyo F., M.A. Díaz and Castellanos J.:* **Hierarchical Knowledge Representation: Symbolic Conceptual Trees and Universal Approximation**. *International Journal of Intelligent Control and Systems (IJICS). ISSN: 0218-7965 Vol.: 12. Issue: 2. Pp.: 142–149. 2007.*

9. *Miguel Angel Díaz, Luis Fernando de Mingo López, Nuria Gómez Blas:* **Networks of Evolutionary Processors: Java Implementation of a Threaded Processor**. *Information Research and Applications. Vol.: 1. ISSN:1313-1109. Pp.: 203-210. 2007.*

10. *Miguel Angel Peña ,Miguel Angel Díaz, Luis Fernando de Mingo López:* **Networks of Evolutionary Processors UML Architecture**. *The 5th WSEAS international conference on Computation Intelligence Man, Machine System and Cybernetic (CIMMACS 06) November 20-22 ISBN: 960-8457-56-4. Pp: 176-183 Venecia Italia 2006*

11. *N. Gómez Blas, Eugenio Santos y Miguel Angel Díaz:* **Symbolic Learning(clustering) Over DNA String**. *WSEAS Press ISSN:1790-0832. Volumen 4 Pp.: 617-624. 2007 Venecia Italia.*

12. *Miguel Angel Díaz, Miguel Angel Peña, Luis Fernando de Mingo López:* **Simulation of Networks of Evolutionary Processors with Filtered Connection**. *WSEAS Press ISSN:1790-0832. Volumen 4 Pp.: 608-616. 2007 Venecia Italia.*

13. *Miguel Angel Díaz, Nuria Gómez Blas, Eugenio Santos, Rafael Gonzalo y Francisco Gisbert :* **Networks of Evolutionary Processors and Decision Support System**. *Vth International Conference Information Research and Aplicaction ITech 2007 ISSN:1313-1109. Volumen 1 Pp.: 197-202. 2007 Varna Bulgary*

14. *M. Angel Diaz, L.F. de Mingo, N. Gómez Blas and J. Castellanos:* **Implementation of Massive Parallel Networks of Evolutionary Processors (MPNEP): 3-Colorability Problem.** *International Workshop on Nature Inspired Cooperative Strategies for Optimization. Acireale, Sicily (Italy), November 8-10, 2007 Studies in Computational Intelligence, Springer. ISSN: 1860-949X. 2008.*

15. *Miguel Angel Díaz, Luis Fernando de Mingo López, Nuria Gómez Blas:* **Networks of Evolutionary Processors: Java Implementation of a Threaded Processor**. *Information Research and Applications. Vol.: 15. ISSN: 1313-1109. Pp.: 37-43. 2008.*

16. *Luis Fernando de Mingo López, Nuria Gómez Blas,Miguel Angel Díaz* **A String Measure with Symbols Generation: String Self-Organizing Maps** *15th International Conference, ICONIP NOV 2008*